6-1-2009

Optimizing scalar multiplication for koblitz curves using hybrid FPGAs

Gregory Gluszek

Optimizing Scalar Multiplication for Koblitz Curves Using Hybrid FPGAs

by

Gregory A. Głuszek

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Engineering

Supervised by

Dr. Marcin Łukowiak
Professor, Department of Computer Engineering
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
June 2009

Approved By:

Dr. Marcin Łukowiak
Professor, Department of Computer Engineering

Dr. Stanisław P. Radziszowski
Professor, Department of Computer Science

Dr. Muhammad Shaaban
Professor, Department of Computer Engineering

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: Optimizing Scalar Multiplication for Koblitz Curves Using Hybrid FPGAs

I, Gregory A. Głuszek, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or part.

Gregory A. Głuszek

Date

Dedication

To my family.

Acknowledgments

I would like to acknowledge my committee, Dr. Marcin Łukowiak, Dr. Stanisław Radziszowski and Dr. Muhammad Shaaban, for their support throughout the process of researching and implementing the work done in thesis. Specifically I would like to thank Dr. Łukowiak for guiding me towards this area of research in the first place and Dr. Radziszowski for teaching me what I know about cryptography. I would also like to thank Glenn Ramsey Jr. for allowing me to use the normal basis multiplier he implemented for the work done in his thesis and for helping me through the initial caveats of learning how to use the capabilities of the Virtex-4 FX60.

Abstract

Elliptic curve cryptography (ECC) is a type of public-key cryptosystem which uses the additive group of points on a nonsingular elliptic curve as a cryptographic medium. Koblitz curves are special elliptic curves that have unique properties which allow scalar multiplication, the bottleneck operation in most ECC cryptosystems, to be performed very efficiently.

Optimizing the scalar multiplication operation on Koblitz curves is an active area of research with many proposed algorithms for FPGA and software implementations. As of yet little to no research has been reported on using the capabilities of hybrid FPGAs, such as the Xilinx Virtex-4 FX series, which would allow for the design of a more flexible single-chip system that performs scalar multiplication and is not constrained by high communication costs between hardware and software.

While the results obtained in this thesis were competitive with many other FPGA implementations, the most recent research efforts have produced significantly faster FPGA based systems. These systems were created by utilizing new and interesting approaches to improve the runtime of performing scalar multiplication on Koblitz curves and thus significantly outperformed the results obtained in this thesis. However, this thesis also functioned as a comparative study of the usage of different basis representations and proved that strict polynomial basis approaches can compete with strict normal basis implementations when performing scalar multiplication on Koblitz curves.

Contents

List of Figures

List of Tables

Glossary

A

ABC anomalous binary curve

APU Auxiliary Processor Unit

B

BRAM block RAM

E

EEC Elliptic Curve Cryptography

EEA Extended Euclidian Algorithm

F

FCM Fabric Co-processor Module

FF flip-flop

G

GNB Gaussian Normal Basis

L

LD Lopez and Dahab

LCBP Low Complexity Bit Parallel

LUT lookup table

N

NIST National Institute of Standards and Technology

NAF non-adjacent form

P

PPC PowerPC

R

RIT Rochester Institute of Technology

RTNAF reduced τ-adic non-adjacent form

RTNAF$_w$ reduced width-w τ-adic non-adjacent form

T

TNAF τ-adic non-adjacent form

U

UDI User Defined Instruction

Chapter 1

Introduction

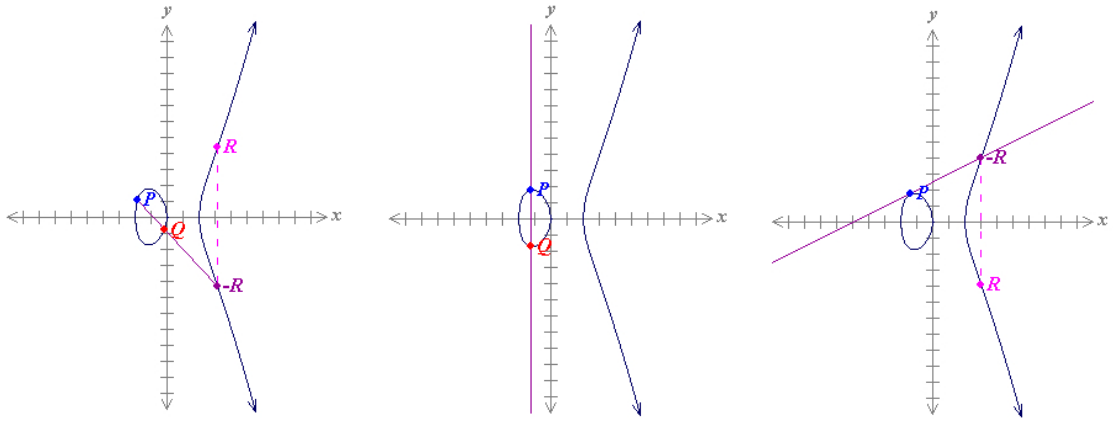
1.1 Elliptic Curve Cryptography

Cryptography is the study and practice of encoding sensitive information such that only intended recipients are able to decode and understand it. There are two primary categories for cryptographic systems, asymmetric key systems, also known as public-key cryptosystems, and symmetric key systems, also known as private-key cryptosystems. With symmetric key systems two users share an identical key, which is used for both encoding and decoding data. With asymmetric key systems each user has their own unique public and private key pairs, thus eliminating the need for, or providing means to, establish key agreement.

Elliptic curve cryptography is a public-key cryptosystem that has been gaining popularity for the past several years. Unlike many other public-key cryptosystems, such as RSA, which requires keys to be in the thousands of bits in order to be secure, elliptic curve cryptography is able to provide secure data transactions with keys sized in the hundreds of bits. For example an RSA system using a 1024-bit key has comparable security to an elliptic curve cryptosystem using a 163-bit key [3].

An elliptic curve is defined as the set of points $P = (x, y)$ which are solutions to the bivariate cubic equation, commonly known as the Weierstrass equation, defined over a field K:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \text{ where } a_i \in K \tag{1.1}$$



(a) $P + Q = R$ (b) $P + Q = \mathcal{O}$ (c) $P + P = R$

Figure 1.1: Point Addition on $y^2 = x^3 - 4x$

In the case of elliptic curve cryptography, we are interested in nonsingular elliptic curves, which have non-zero discriminants, defined over the integers modulo a prime, $\mathbb{Z}_p$, or over binary Galois fields, $\mathbb{F}_{2^m}$.

The points on one of these curves, along with a special point called the point at infinity, $\mathcal{O}$, can be viewed as an abelian group with $\mathcal{O}$ functioning as the neutral element. The operation over which one of these groups is defined is denoted as point addition. Point addition can be conceptualized as the operation of finding the third intersection of a line that intersects the two points being added on an elliptic curve. Figure 1.1 shows this graphically for the elliptic curve $y^2 = x^3 - 4x$, which is defined over the real numbers. A group formed by the points on an elliptic curve and defined under point addition is either cyclic or contains a subgroup that is cyclic and functions as the medium for ECC cryptosystems. Scalar multiplication, denoted kP, where k is a scalar and P is a point on a particular elliptic curve, is the primary operation in most ECC cryptosystems. This operation consists of adding the point P to itself $k - 1$ times and is often the computational bottleneck in ECC cryptosystems.

1.2 Koblitz Curves

Koblitz curves, also known as *anomalous binary curves* (ABCs), are a special type of elliptic curve defined over $\mathbb{F}_{2^m}$. There are two Koblitz curves in any given field, denoted E_0 and E_1, and they are defined by the equation:

$$E_a : y^2 + xy = x^3 + ax^2 + 1 \text{ where } a \in \{0, 1\} \tag{1.2}$$

These curves were first introduced by Neal Koblitz in [11]. In [22] Solinas introduced improved algorithms for computing scalar multiples on these curves.

1.2.1 The Frobenius Map

Koblitz curves are of particular interest in that they have several properties which allow scalar multiplication to be computed much more efficiently that on generic elliptic curves. In the case of generic elliptic curves, kP can be efficiently computed using a technique known as double-and-add. The idea behind the double-and-add approach is that rather than naively adding a point to itself k times, the kP operation can be broken down into a sequence of point additions ($P + Q$, where $P \neq Q$) and point doublings (where $P + P$ can be calculated in an efficient manner because we know that the two points are identical). The cost of the double-and-add method is broken down such that the number of point doublings is based on the bit length of k and the number of point additions is based on the Hamming weight of k's binary representation. For example, if we wish to compute kP where $k = 27$, we know that :

$$27_{10} = 11011_2 = 2(2(2(2 + 1)) + 1) + 1 \tag{1.3}$$

Therefore, with 4 point doublings and 3 point additions we can compute $27P$ as:

$$2(2(2(2P + P)) + P) + P = 27P \tag{1.4}$$

In the case of Koblitz curves the point doublings in the double-and-add routine can be replaced with a much cheaper operation known as the Frobenius map, denoted τ. Applications of the Frobenius map simply consist of squaring the coordinates of a given point.

That is, given $P = (x, y)$, $\tau P = (x^2, y^2)$.

1.2.2 Scalar Recoding

In order to take advantage of the Frobenius map when computing kP, the scalar k must be recoded in base τ. There are numerous ways to recode k in base τ. In this thesis we will discuss three such methods, τ-adic non-adjacent form (TNAF), reduced τ-adic non-adjacent form (RTNAF) and reduced width-w τ-adic non-adjacent form (RTNAF$_w$), with TNAF being an inefficient option and RTNAF being a specific instance of RTNAF$_w$. But first, in order to recode k in base τ we must have a numerical value for τ. If we define $\mu = (-1)^{1-a}$, where a designates which of the two Koblitz curves we are working with for a particular field, then $\tau = \frac{\mu+\sqrt{-7}}{2}$. The value of τ is based on the fact that $(x^4, y^4) + 2(x, y) = \mu \cdot (x^2, y^2)$ holds true for any point on a Koblitz curve E_a [22]. By recoding k in base τ and keeping in mind that τ is both a complex number and the Frobenius map operation, scalar multiplication on a Koblitz curve can simply be viewed as a series of independent point additions and applications of the Frobenius map. For example, if we are working on the Koblitz curve E_1 and we wish to compute $27P$, we must first recode 27 in base τ:

$$27_{10} = 10011010111_\tau = \tau^{10} + \tau^7 + \tau^6 + \tau^4 + \tau^2 + \tau + 1 \tag{1.5}$$

Then by redistributing point P amongst the base τ representation of 27 and subsequently switching our viewpoint of τ from a complex number to the Frobenius map, we can compute $27P$ with 6 point additions and 6 applications of the Frobenius map:

$$\tau^{10}P + \tau^7 P + \tau^6 P + \tau^4 P + \tau^2 P + \tau P + P = 27P \tag{1.6}$$

TNAF

Since the number of point additions is based on the Hamming weight of the scalar k's representation, by utilizing methods to reduce the Hamming weight the runtime of the scalar multiplication process can be decreased significantly. The first such method would

be to recode the scalar in τ-adic non-adjacent form:

$$\text{TNAF}(k) = \sum_{i \leq 0}^{l-1} u_i \tau^i, \text{ where } u_i \in \{-1, 0, 1\} \text{ and } u_i \cdot u_{i+1} = 0 \tag{1.7}$$

For example:

$$\text{TNAF}(27) = \langle 1, 0, 1, 0, 0, -1, 0, 0, 0, -1, 0, -1 \rangle_\tau \tag{1.8}$$

This allows us to compute 27P with 4 point additions and 4 applications of the Frobenius map:

$$\tau^{11}P + \tau^9 P - \tau^6 P - \tau^2 P - P = 27P \tag{1.9}$$

The downside to recoding k in TNAF is that this method gives a representation with an average Hamming weight of approximately $\frac{2}{3}\log_2(k)$ [22]. This is a rather inefficient choice for recoding as the average Hamming weight of binary representation is $\frac{1}{2}\log_2(k)$.

RTNAF

TNAF recoding of k can further be reduced for a particular Koblitz curve in a given field. We call such a method reduced τ-adic non-adjacent form and define it as:

$$\text{RTNAF}(k) = \text{TNAF}(k) \bmod \delta, \text{ where } \delta = (\tau^m - 1)/(\tau - 1) \tag{1.10}$$

RTNAF(k) results in an average Hamming weight of approximately $\frac{1}{3}\log_2(k)$ [22]. The basis of computing TNAF(k) mod δ is that this ensures that the point is reduced to being within the *main subgroup*, a cryptographically secure subgroup of E_a [22]. Since the NIST recommended curves always require us to work within the *main subgroup*, RTNAF(k) is always a valid method for recoding k.

RTNAF$_w$

Finally, the average Hamming weight of the scalar k's representation can be even further decreased by adapting a window method and using what is known as reduced width-w

TNAF recoding. This is done by computing:

$$\text{RTNAF}_w(k) = \sum_{i \leq 0}^{l-1} u_i \tau^i \bmod \delta \tag{1.11}$$

Again $u_i \cdot u_{i+1} = 0$, however, now each u_i is an odd number that is less than 2^{w-1} in absolute value. This further reduces the average Hamming weight to approximately $\frac{1}{w+1}\log_2(k)$ [22]. It should be noted that for $w = 2$, RTNAF$_w$ recoding is equivalent to RTNAF recoding. Also, it is important to note that the downside of RTNAF$_w$ recoding is that as w increases so does memory usage. In order for RTNAF$_w$ recoding to be effective, odd point multiples up to 2^{w-2} must be readily available. One common method is to precompute these values and store them in memory. According to [23], for Koblitz curves, a width of $w = 4$ or $w = 5$ is ideal as it offers an optimal tradeoff between decreasing the Hamming weight of RTNAF$_w(k)$ and minimizing the number of point multiples that must be readily available.

Additionally it is important to mention that other techniques are available to further reduce the Hamming weight of k's representation, such as those presented in [2] and [12]. However, these techniques offer little improvement in further reducing the average Hamming weight and, therefore, will not be studied in the context of this thesis.

1.3 Hybrid FPGAs

Hybrid FPGAs, such as Xilinx's Virtex-4 FX series, are unique devices in which general purpose processors are implemented in an FPGA's fabric so as to provide a medium in which a mixed software and hardware approach can be utilized with little communication overhead. In the case of the Virtex-4 FX series, two PowerPC (PPC) processor cores are embedded in the FPGA fabric. The Virtex-4 FX series not only allows for portions of algorithms to be implemented in software as well as hardware, but also supports User Defined Instructions (UDIs) so that hardware units can easily be accessed by software. This allows a system created on this FPGA to be much more versatile, as hardware units

can easily be exchanged, rearranged or used in a different manner with minimal design overhead.

Chapter 2

Thesis Objectives

The objective of this thesis is to utilize the capabilities of a hybrid FPGA to produce a system in which the scalar multiplication operation is optimized for Koblitz curves. When performing scalar multiplication on Koblitz curves there are certain operations that are better suited for hardware implementations (i.e. finite field arithmetic) and others better suited for software implementations (i.e. scalar recoding). Overall the goal of this thesis is to utilize the capabilities of a hybrid FPGA, such as the Virtex-4 FX60, and attempt to develop a scalar multiplication unit that is better than other software and hardware implementations, such as those presented in [23], [16] and [15].

2.1 Supporting Work

One notable research effort in the area of performing scalar multiplication on Koblitz curves is the FPGA based elliptic curve co-processor design presented in [14]. In this paper a co-processor architecture, meant to be used in conjunction with software running on a separate chip, is presented and implemented on a Xilinx XCV2000E FPGA. This design is optimized to perform certain portions of the scalar multiplication operation on Koblitz curves. The co-processor utilizes the reduced τ-adic non-adjacent form (RTNAF) representation of the scalar k, where RTNAF(k) is given as an input to the co-processor. It is important to note that RTNAF is not a commonly used number representation nor is conversion from binary to RTNAF a trivial process. Lopez and Dahab projective coordinates are utilized to

reduce inverse calculations and the polynomial basis is utilized to represent the underlying field of the given curve. Surprisingly no mention is made of normal basis representation even with the large number of repetitive squarings that are required when using RTNAF. Itoh and Tsujii's method [6] is used to calculate field inverses and a detailed argument is given as to why this method was chosen as opposed to the Extended Euclidean Algorithm (EEA). This paper has many interesting ideas and equally interesting results, yet leaves many other possibilities to be explored.

Another notable research effort is the circuitry for TNAF recoding presented in [7]. This paper claims to be the first to present a hardware design for implementing TNAF recoding. While the paper proposes an interesting answer to the question of how it is best to perform scalar recoding, which was left unanswered by research efforts such as those presented in Lutz's M.S. Thesis [14], this paper only deals with RTNAF recoding and does not look into width-w RTNAF. Width-w RTNAF, known also as the windowing method, can significantly reduce the Hamming weight of a representation of scalar k even further than normal RTNAF. The authors in [7] do provide good ideas of how to implement RTNAF conversion in hardware and show what can be expected from such implementations. However, certain improvements to RTNAF recoding that are arguably necessary for efficient scalar multiplication on Koblitz curves are left unmentioned.

The most notable and recent research related to improving the scalar multiplication operation on Koblitz curves is the Short-Memory approach presented in [23]. In this paper Vuillaume *et al.* present hardware and software algorithms for scalar multiplication that take advantage of the width-w τ-adic non-adjacent form (TNAF$_w$) scalar recoding technique, but bypass the requirement of storing 2^{w-2} precomputed points. Vuillaume *et al.* accomplish this by utilizing the commutative and associative properties of point addition in order to compute and add point multiples in increasing order, rather than in the order given by their TNAF$_w$ representation. For the hardware implementation a normal basis representation is proposed, while a mixed basis approach is used in software. Additionally, Lopez and Dahab coordinates are used to reduce the number of field inverses that must be

calculated.

While Vuillaume *et al.* present a new and intuitive way to improve scalar multiplication on Koblitz curves they still leave many areas open for further research. First of all, while they explore both a hardware and software approach, they mention nothing of a mixed approach or any comparison between their hardware and software algorithms. Second, by noticing and taking advantage of the commutative and associate properties of point addition, Vuillaume *et al.* work from a completely new angle, but they only take this idea so far. Vuillaume *et al.* focus solely on minimizing memory usage, while if the memory constraints were relaxed many new and interesting ideas could be explored. For example, in order to save on memory the scalar k is recoded several times, as well as iterated over several times, during the scalar multiplication operation. It is argued that the computational cost of this operation is negligible and that significant memory is being saved. It would seem that if more memory were used, the recoded k could be stored, and even possibly sorted according to point multiple sizes, so that not only would k have to be recoded only once, but also so that only one iteration over the recoding of k would be required. Another improvement that was not mentioned is parallelization. When recoding the scalar k in TNAF_w, scalar multiplication consists of independent applications of the Frobenius map and point additions. If the point addition and Frobenius mapping units were small enough, and the implementation medium large enough, multiple units could be created to parallelize scalar multiplication.

Other papers, such as [16], [21] and [5], also provide software and hardware architectures for scalar multiplication, however, they do not focus on Koblitz curves. These papers function as good resources for understanding what has been done to improve scalar multiplication on generic curves, and give ideas as to what underlying architectures are most suitable. However, they lack information on many of the finer issues which are unique to Koblitz curves.

2.2 Thesis Details

While the objectives of this thesis have already been stated at a high level, it is important to outline certain details. First of all, this work focuses solely on the five Koblitz curves that have been recommended by NIST in [17] for their efficiency and cryptographic strength. These curves can be recognized by the fields they are defined over, which are $\mathbb{F}_{2^{163}}$, $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$ and $\mathbb{F}_{2^{571}}$.

Second, this thesis will also look at and compare the usage of different basis representations for the underlying finite fields. Finite fields are most often represented using a polynomial basis, as it is intuitive and there are algorithms and architectures for fast arithmetic in both hardware and software. However, using a normal basis representation is also a popular approach as it results in squaring that can be computed for free in hardware. Additionally, at times a mixed basis is used, taking advantage of the fast squaring of normal basis elements, while not losing the quick multiplication of polynomial basis elements. These three approaches will be explored as part of this thesis.

Finally, the polynomial basis and normal basis multiplier that will be used in this thesis are both digit-width variable. This means that a multiplication on the field $\mathbb{F}_{2^m}$ can be computed in 1 to m clock cycles, with faster multipliers requiring additional resources. The efficiency of various digit widths for the normal basis and polynomial basis multipliers will be studied as part of this thesis. In addition, the basis conversion unit, used to convert from polynomial basis to normal basis and vice versa, is also digit width variable and will be studied in a similar manner.

Chapter 3

Implementation Choices

Implementing scalar multiplication on Koblitz curves requires many important decisions. This includes the high level architectural decisions, such as division of the scalar multiplication algorithm into software processes and hardware units, as well as the specifics of the techniques and architectures used for implementing each software process and hardware unit. Figure 3.1 shows the data flow for the scalar multiplication algorithm for Koblitz curves at a high level while Algorithm 1 shows the details of the algorithm. The primary operations of concern are scalar recoding, the Frobenius map and point addition.

Another important detail is that each of the five NIST recommended Koblitz curves is defined over a finite field of the form $\mathbb{F}_{2^m}$ and requires the squaring, multiplication, addition and inversion of $\mathbb{F}_{2^m}$ elements. The elements of a binary Galois field can be represented in different ways, with polynomial basis and normal basis being the most common in the context of ECC. The choice of basis representation significantly affects the implementation of the finite field arithmetic, and for this reason three distinct implementations styles were examined. These styles can be categorized as a polynomial basis approach, a normal basis approach and a mixed basis approach. The names of these styles correspond to the method chosen for implementing the underlying finite field arithmetic, with the mixed basis approach taking advantage of computationally negligible squaring in normal basis representation and comparatively fast multiplication in polynomial basis representation.

The following sections outline the details of each of the three styles, detailing the choice

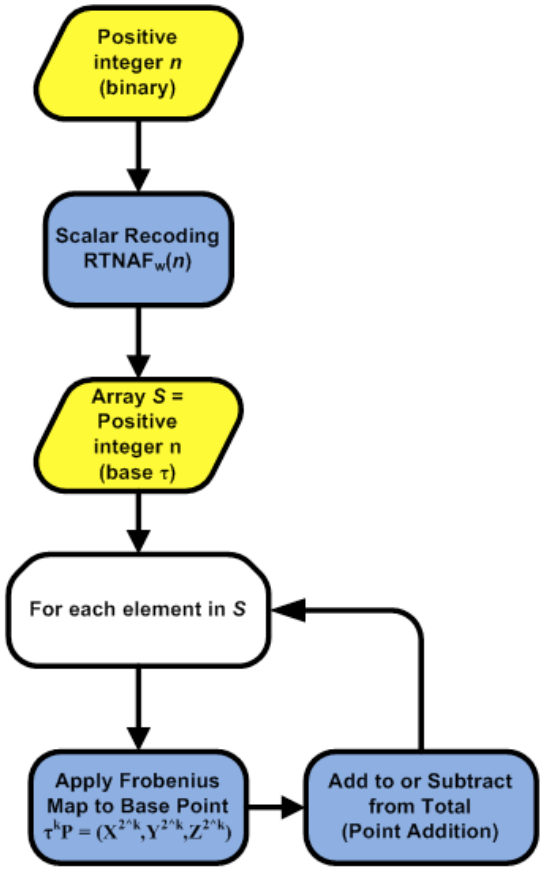


Figure 3.1: Data Flow Diagram for Scalar Multiplication on Koblitz Curves

Algorithm 1 Scalar Multiplication on Koblitz Curves Utilizing RTANF$_w$ [4]

INPUT: k, a positive integer less than $r/2$ (where r is the order of the *main subgroup*)
P, a point in the *main subgroup*
w, the window width
OUTPUT: The point kP
Compute RTNAF$_w(k) = \sum_{i=0}^{l-1} u_i \tau^i$ (See Algorithm 2)
Compute $P_u = \alpha_u P$, for $u \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$
$Q \Leftarrow \mathcal{O}$
for i from $l - 1$ downto 0 **do**
  if $u_i \neq 0$ **then**
    Let u be such that $\alpha_u = u_i$ or $\alpha_{-u} = -u_i$
    if $u > 0$ **then**
      $Q \Leftarrow Q + \tau^i P_u$
    else
      $Q \Leftarrow Q - \tau^i P_{-u}$
    end if
  end if
end for

of a hardware versus a software approach for each major operation in the scalar multiplication algorithm and the corresponding techniques and architectures used to implement the specific operation. Each section also details the overall architecture used to realize an entire scalar multiplication unit. Since the methods for performing scalar recoding, point addition, finite field element inversion and finite field element addition are independent of the representation of $\mathbb{F}_{2^m}$ elements, they have been mentioned separately.

3.1 System Architecture

Before delving into the specifics of the implementation choices, it is important to understand the details and capabilities of the platform that will be used. One of the key features of the Virtex-4 FX60 FPGA is the inclusion of two PowerPC 405 processors in the FPGA fabric. This section presents a high level view of the system architecture, specifically focusing on the interface between the FPGA and PPC cores and the design choices related to best utilizing the capabilities of the Virtex-4 FX60. Further details on the Virtex-4 FX60

architecture can be found in [1].

3.1.1 The APU and FCM

Figure 3.2 gives a high level view of the communication interface between the PPC and FPGA. The PPC comes equipped with an Auxiliary Processing Unit (APU) controller which decodes a specific set of assembly instructions. The APU is connected to the FPGA and, in the case of this thesis, communicates directly with a custom Fabric Co-processor Module (FCM) which was created to process APU decoded instructions. In other words the FCM used in this thesis is essentially a control unit which, depending on the instruction decoded by the APU, either sends or receives data (load/store instructions), or activates particular hardware units with specified parameters (User Defined Instructions).

3.1.2 Instruction Classes

The FCM created for this thesis can handle two types of instructions decoded by the APU. Load/store instructions allow for up to four 32-bit words of data to be exchanged between the FPGA and PPC in a single instruction, while User Defined Instructions (UDIs) signal the hardware to perform specific operations on the data stored in the FPGA, such as Galois field arithmetic. When a load instruction occurs either one, two or four 32-bit words are sent from the PPC to the FPGA and stored in registers. When a store instruction occurs either one, two or four 32-bit words are sent from the FPGA to the PPC.

Xilinx defines three different instruction classes which can be used to specify how a UDI will execute and subsequently define how the PPC will operate when executing a particular UDI. The first class is Autonomous Instructions. Autonomous Instructions are instructions which do not stall the PPC pipeline once execution has begun and have no return values. The other two classes of instructions are defined as Non-autonomous Instructions, as they do stall the execution of the PPC pipeline after execution has begun and

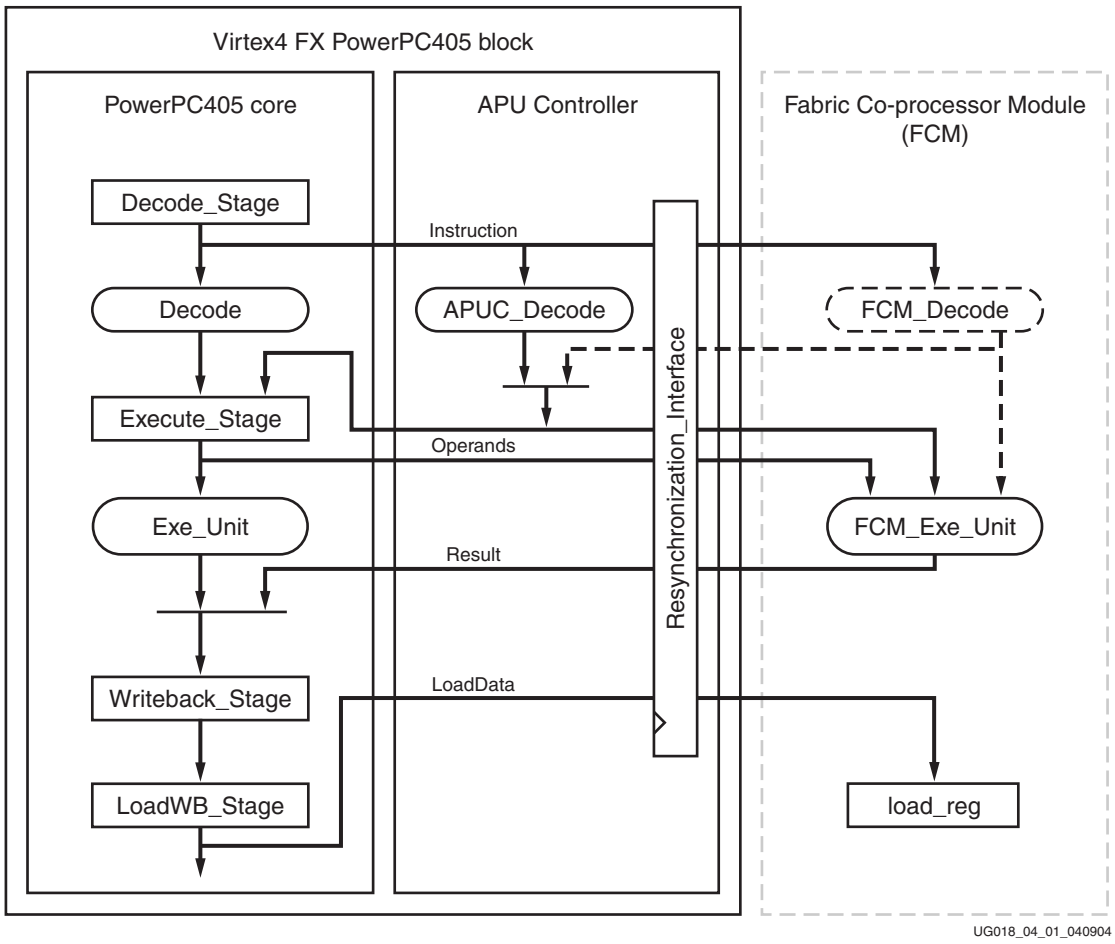


Figure 3.2: Pipeline Flow Diagram [1]

have return values. Non-autonomous Instructions fall into one of two groups, Blocking Instructions or Non-blocking Instructions. Blocking Instructions are instructions that cannot be predictably aborted and later reissued and, therefore, block interrupts and exceptions once they have begun to execute. Non-blocking Instructions, on the other hand, can be aborted and later re-issued if needed. In the case of this thesis all UDIs are implemented as Autonomous Instructions as there are no return values. Results of all UDIs are registered in the FPGA and data transactions between the FPGA and PPC are accomplished solely using load/store instructions.

3.2 Scalar Recoding

As mentioned in the Introduction, in order to fully take advantage of the special properties of Koblitz curves when computing kP, scalar k must be recoded in RTNAF$_w$. There are two primary methods to take into consideration when deciding how to implement the recoding. The first is that proposed by Solinas in [22] in which reduction occurs before the actual recoding. The second is the approach which was proposed and utilized by Lutz in [14] where reduction is performed after the recoding phase. Lutz's approach is attractive in that it does not require multiplication or division of very large numbers (those greater than 128 bits in width). However, Lutz's approach solely focuses on RTNAF$_w$ with a width of $w = 2$ and does not provide a method for calculating RTNAF$_w$ with larger widths. There is also the fact that Solinas's reduction method is more refined than Lutz's method as Lutz reduces by $(\tau^m - 1)$ where Solinas reduces by $\delta = (\tau^m - 1)/(\tau - 1)$. The difference between these two reductions is that Lutz's method reduces k based upon the size of $\#E$, the number of points on elliptic curve E, while Solinas refines the reduction by reducing k based upon the size of what he refers to in [22] as the *main subgroup* . The *main subgroup* is a subgroup of $\#E$ of order r, where r is a prime number. Since we will always be working in the *main subgroup*, as it provides cryptographic strength and is therefore recommended by NIST, Solinas's reduction method will always be applicable. Due largely to the fact that Solinas's

method is much more refined than Lutz's, the method presented by Solinas in [22] was utilized to implement RTNAF$_w$. Algorithm 2 shows Solinas' method for RTNAF$_w$ recoding (details on the process of computing n mod δ can be found in [22]).

Algorithm 2 Reduced width-w τ-adic non-adjacent form [22]

INPUT: A positive integer n
OUTPUT: RTNAF$_w(n)$
  $(r_0, r_1) \Leftarrow n$ mod δ
  while $r_0 \neq 0$ or $r_1 \neq 0$ **do**
    if (r_0 mod $2 == 1$) **then**
      $u \Leftarrow r_0 + r_1 t_w$ mods 2^w
      $r_0 \Leftarrow r_0 - u$
    else
      $u \Leftarrow 0$
    end if
    Prepend u to S
    $(r_0, r_1) \Leftarrow (r_1 + \mu r_0/2, -r_0/2)$
  end while
  Output S

Solinas's method for computing RTNAF$_w$ is inherently sequential, and requires the addition, subtraction, multiplication and division of large numbers (greater than 128 bits), and therefore was implemented entirely in software. Addition and subtraction were performed using two's complement numbers stored in arrays of 32-bit integers. Multiplication was performed with unsigned numbers of variable length, so as to speed up the frequent occurrence of multiplying large numbers by small numbers. Division was avoided altogether by utilizing pre-computed values.

In Solinas's original formulas only two divisions need to be computed for each RTNAF$_w$ conversion. These divisions take place in the reduction phase (computing n mod δ). Solinas proposed a partial modular reduction formula which replaced the divisions with a few additional multiplications, additions and bit-shifts, essentially utilizing fixed point arithmetic. The downside to the partial modular reduction formula is that although it results in reductions that are accurate, they are not necessarily of minimal size. Since the divisions in question are actually of the form $s_i k/r$, where s_i and r are constants, the division of s_i

by r were pre-computed in fixed point and the division step was avoided altogether. This resulted in the equivalent of utilizing partial modular reduction, but without requiring as many instructions.

Another important issue to mention is the adoption and modification of ideas presented in [23]. Normally when utilizing NAF$_w$ with a width greater than 2, point multiples up to 2^{w-2} must be readily available, and often are pre-computed and stored in memory. However, in the case of Koblitz curves where RTNAF$_w$ is utilized, the computation of kP becomes a series of independent point additions and applications of the Frobenius map. Due to the commutative property of point addition, as the points on an elliptic curve form an abelian group, the RTNAF$_w$ representation of k can be reordered so that point multiples are arranged from smallest to largest. It is important to note that this reordering can take place during the recoding process so that no additional time is used iterating over RTNAF$_w(k)$ multiple times or sorting the non-zero values after recoding. This results in a system that is only slightly less memory efficient than when using a width of $w = 2$, as the point $2P$ must be stored in memory to update the base point, and is only slightly slower than when using pre-computed points, as the base point must be updated $2^{w-2} - 1$ times. The average number of point additions required for each reasonable width and NIST recommended field size is shown in Table 3.1. This is based upon the fact that RTNAF$_w$ has an average hamming weight of $\frac{1}{(w+1)}\log_2(k)$ and will require an additional $2^{w-2} - 1$ point additions in order to update the base point.

Field Size	**RTNAF$_2$**	**RTNAF$_3$**	**RTNAF$_4$**	**RTNAF$_5$**	**RTNAF$_6$**	**RTNAF$_7$**
$m = 163$	55	42	36	**35**	39	52
$m = 233$	78	60	50	**46**	49	61
$m = 283$	95	72	60	**55**	56	67
$m = 409$	137	104	85	76	**74**	83
$m = 571$	191	144	118	103	**97**	103

Table 3.1: Average Point Additions Required with RTNAF$_w$

3.3 Point Addition

The process of adding two points on an elliptic curve can vary significantly depending on the coordinate system used to store individual point data. Affine coordinates are a simple and intuitive method of representing elliptic curve points in which a point is represented as $P = (x, y)$, where $x, y \in \mathbb{F}_{2^m}$. When using affine coordinates very few finite field multiplications and squarings are required per point addition, however, this comes at the cost of computing a field inverse for each point addition. In contrast to affine coordinate representation there is projective coordinate representation in which point coordinates are stored as triples where $P = (X, Y, Z)$, and $X, Y, Z \in \mathbb{F}_{2^m}$. The primary benefit of projective coordinates is that no field inversions are required when adding points together. However, the use of projective coordinates does come at the cost of additional finite field multiplications and squarings. The tradeoff between using projective coordinates or affine coordinates comes down to the cost ratio of finite field inversions versus finite field multiplications and squarings.

There are a number of different types of projective coordinates and the differences can most clearly be seen in the conversion process back to affine coordinates. The different projective coordinates also affect the point addition formula and certain projective coordinates are more efficient than others. Typically the conversion from affine to projective coordinates is of the form $(x, y) = (X/Z^a, Y/Z^b)$ where a and b vary depending on the type of projective coordinates. In the case of this thesis Lopez and Dahab (LD) projective coordinates were chosen as they are the most efficient since they require the fewest number of squarings and multiplications per point addition [13]. This can be seen in Table 3.2 which compares three projective point coordinate representations in terms of the number of squarings and multiplications required per point addition. The operation being computed for Table 3.2 is $(X_0, Y_0, Z_0) + (X_1, Y_1, 1)$. Figure 3.3 shows the algorithm for computing $(X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2)$ using LD coordinates. The algorithm has been represented in a flow diagram to show the data dependencies.

Although finite field inversions are not required when adding points represented in LD

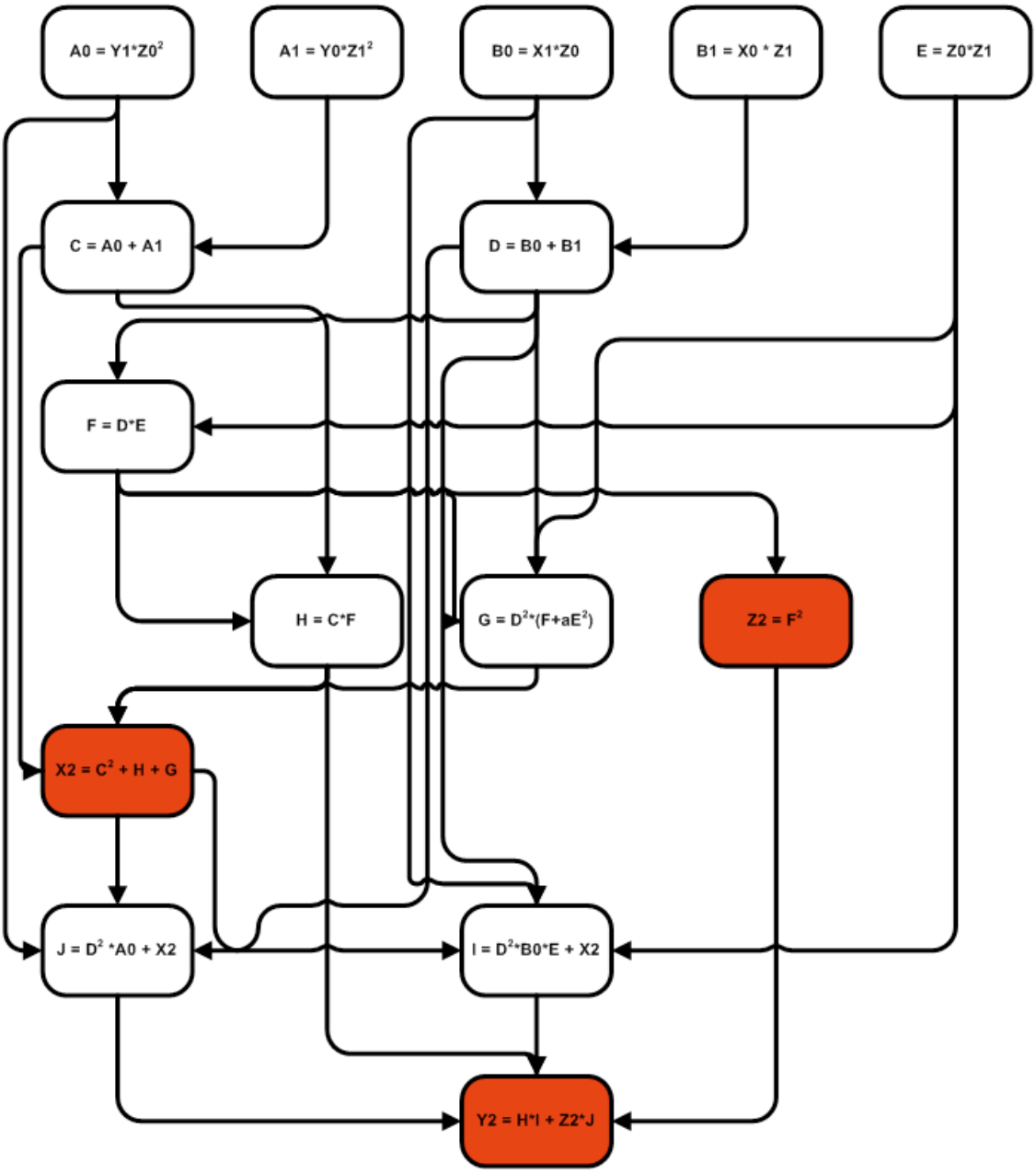


Figure 3.3: $(X_0, Y_0, Z_0) + (X_1, Y_1, Z_1) = (X_2, Y_2, Z_2)$ Using LD Projective Coordinates

coordinates, they still must be calculated when converting back to affine coordinates. While LD coordinates were chosen for their efficiency in the case of this thesis, they are not as ubiquitous as affine coordinates. Therefore, the capability to convert from LD coordinates to affine is a necessity.

Finally, in the case of point addition on Koblitz curves, LD coordinates have an additional downside when compared to affine. This downside is that the identity element $\mathcal{O}$ exists only in theory. When using affine coordinates on a Koblitz curve, the point $\mathcal{O}$ can be represented as $\mathcal{O} = (0, 0)$ such that when using the formula for point addition $P + (0, 0) = P$. In the case of LD coordinates the point $\mathcal{O}$ has no mathematical equivalent, so special cases must be added into the point addition formula to check for the case of $P + \mathcal{O}$. While this does speed up point addition in the case of $P + \mathcal{O}$, this slows down point addition in all other cases as unnecessary conditional statements must be evaluated.

Name	**Conversion**	**Doubling**	**Addition**
Standard	$(X/Z, Y/Z)$	$7M + 5S$	$12M + 1S$
Jacobian	$(X/Z^2, Y/Z^3)$	$5M + 5S$	$10M + 4S$
Lopez and Dahab	$(X/Z, Y/Z^2)$	$4M + 5S$	$9M + 4S$

Table 3.2: Projective Coordinates Comparison (M = multiplication, S = squaring) [13]

3.4 Finite Field Inversion

There are three popular methods for computing the inverse of a finite field element: the Extended Euclidian Algorithm (Algorithm 3), Itoh and Tsujii's method (Algorithm 4), and addition chains (Algorithm 5). In the case of polynomial basis elements it might at first seems obvious that the Extended Euclid Algorithm (EEA) should be utilized to perform field inversions. However, the EEA requires $2m$ clock cycles on average, which is slower than Itoh and Tsujii's method if a quick enough multiplier is utilized. The requirement of a multiplier is a major downside of Itoh and Tsujii's method and addition chains as compared to EEA. When working with polynomial basis elements EEA can be realized with a simple

hardware configuration that does not require finite field arithmetic units. However, since inversion will never take place at the same time as a multiplication, Itoh and Tsujii's method can use the already present multiplier and, therefore, be just about as resource efficient as modern EEA implementations. Due to the fact that Itoh and Tsujii's method and addition chains can compute inversions just as fast as, and in most cases faster than, EEA and that EEA is only a valid option for elements represented in a polynomial basis, it was concluded that EEA was not an appropriate choice for the work done in this thesis.

Algorithm 3 Inversion in $\mathbb{F}_{2^m}$ using EEA [4]

INPUT: A nonzero binary polynomial $A(x)$ of degree at most $m-1$
OUTPUT: $z \equiv A(x)^{-1} \bmod R(x)$
  $u(x) \Leftarrow A(x)$, $v(x) \Leftarrow R(x)$
  $g_1(x) \Leftarrow 1$, $g_2(x) \Leftarrow 0$
  while $u(x) \neq 1$ **do**
    $j \Leftarrow \deg(u(x)) - \deg(v(x))$
    if $(j < 0)$ **then**
      $u(x) \Leftrightarrow v(x)$
      $g_1(x) \Leftrightarrow g_2(x)$
      $j \Leftarrow -j$
    end if
    $u(x) \Leftarrow u(x) + x^j v(x)$
    $g_1(x) \Leftarrow g_1(x) + x^j g_2(x)$
  end while
  $z \Leftarrow g_1(x)$

The next issue to be addressed is the use of Itoh and Tsujii's method versus addition chains. First of all, in order to aid in understanding Algorithm 5 additional definitions will be provided for addition chains. As defined in [20] addition chains are $U = \{u_0, u_1, \ldots, u_t\}$, with a related sequence $V = \{v_1, v_2, \ldots, v_t\}$ where $v_i = (i_1, i_2)$. It should also be noted that for each $u_i, 1 \leq i \leq t, u_i = u_{i_1} + u_{i_2}$, $u_0 = 1$ and $u_t = m-1$. As can be derived from Algorithms 5 the addition chain method requires t multiplications and $m-1$ squarings per inversion. On the other hand, Algorithm 4 shows that Itoh and Tsujii's method requires $\lfloor \log_2(m-1) \rfloor +$ Hamming weight$(m-1) - 1$ multiplications and $m-1$ squarings. Since addition chains often require several previous partial results to be

Algorithm 4 Itoh-Tsujii Inversion Algorithm [6]

INPUT: $a \in \mathbb{F}_{2^m}$, $m-1 = (m_{l-1}, \ldots, m_1, m_0)_2$
OUTPUT: $z \equiv a^{-1}$
  $z \Leftarrow a^{m_{l-1}}$
  $e \Leftarrow 1$
  for $i = l-2$ downto 0 **do**
    $z \Leftarrow z^{2^e} z$
    $e \Leftarrow 2e$
    if $(m_i == 1)$ **then**
      $z \Leftarrow z^2 a$
      $e \Leftarrow e+1$
    end if
  end for
  $z \Leftarrow z^2$

reused in order to perform field inversions, the decision between these two algorithms boils down to whether or not a short enough addition chain can be formed for $m-1$ for which the additional resource cost is outweighed by the speedup.

Algorithm 5 $\mathbb{F}_{2^m}$ Element Inversion Using Addition Chains [20]

INPUT: $a \in \mathbb{F}_{2^m}$, addition chain U of length t for $m-1$ and its associated sequence V
OUTPUT: $z \equiv a^{-1} \in \mathbb{F}_{2^m}$
  $\beta_0 \Leftarrow a$
  for i from 1 to t **do**
    $\beta_i \Leftarrow (\beta_{i_1})^{2^{u_{i_2}}} \cdot \beta_{i_2}$
  end for
  $z \Leftarrow {\beta_t}^2$

In the end Itoh and Tsujii's method for finite field element inversion was chosen. The reasons for this is that the hardware unit was considerably smaller than the one that performed inversion with addition chains and in the best cases the addition chain inversion formula only reduced the runtime by one or two multiplications. It was determined that the additional runtime for Itoh and Tsujii's method was worth the benefit of a smaller hardware unit.

3.5 Finite Field Addition

The process of adding two binary Galois field elements, whether they are represented in normal basis or polynomial basis, is simply the process of performing a bitwise XOR of the bit strings that represent the two elements. Since implementing this operation in hardware is straightforward, no further discussion will take place.

3.6 Polynomial Basis Approach

Perhaps the most widely used representation for binary Galois field elements is the polynomial basis, which is formed by the set of linearly independent binary vectors $\{x^{m-1}, \ldots, x^2, x, 1\}$, where x^k denotes the single bit that is set in each basis vector. The field is then formed by choosing an irreducible binary polynomial $R(x)$, known as the reduction polynomial, and reducing the set of all binary polynomials modulo $R(x)$. In other words, $GF(2)[x]/R(x)$ represents a binary Galois field with 2^m elements.

The reason that polynomial basis representation is commonly used is because it is intuitive and lends itself to efficient implementations in both hardware and software. The primary argument for the use of normal basis representation is that element squaring is computed as a cyclic shift, which is a free operation in hardware. In most cases it can be argued that this does not necessarily warrant the use of the less intuitive normal basis, as a polynomial basis squaring can easily be computed in a single clock cycle with a very simple and small hardware configuration. However, in the case of Koblitz curves and the utilization of RTNAF$_w$, consecutive element squaring will often need to take place and it is not as clear if a full polynomial basis approach can contend with a normal basis approach in hardware. In an attempt to alleviate this issue the architecture for the polynomial basis approach has been structured to minimize the cost of consecutive squarings.

3.6.1 Polynomial Basis Multiplication

Multiplication in binary Galois fields is fairly intuitive when using a polynomial basis as it can be viewed as the multiplication of two polynomials with binary coefficients followed by reduction modulo $R(x)$. Methods for finite field multiplication of polynomial basis elements can be divided into three categories: bit-serial, bit-parallel and digit-serial.

The Bit-Serial Approach

With bit-serial approach the result is calculated bit by bit in a serial fashion. Algorithm 6 details this approach. Hardware units that implement the bit-serial approach are quick and use minimal resources, however, these units require between m and $2m$ clock cycles to produce a result.

Algorithm 6 Polynomial Basis Bit-Serial Multiplication [4]

INPUT: $A(x)$, $B(x)$ and $R(x)$
OUTPUT: $Z(x) = A(x) \times B(x) \bmod R(x)$
  $Z(x) \Leftarrow 0$
  for $i = m - 1$ downto 0 **do**
    $Z(x) \Leftarrow xP(x) \bmod R(x)$
    if ($b_i == i$) **then**
      $Z(x) \Leftarrow Z(x) + A(x)$
    end if
  end for

The Bit-Parallel Approach

The bit-parallel approach results in a unit that produces a result in a single clock cycle. However, this comes at a price as these architectures are often resource intensive and result in implementations with lower maximum clock frequencies. A good example of such an architecture is Reyhani-Masoleh and Hasan's Low Complexity Bit Parallel (LCBP) multiplier presented in [18] and shown in Figure 3.4. This multiplier breaks polynomial basis multiplication down into three distinct portions, the IP-network, IB and the Q-network. The IP-network computes the multiplication of the two input polynomials, without reduction,

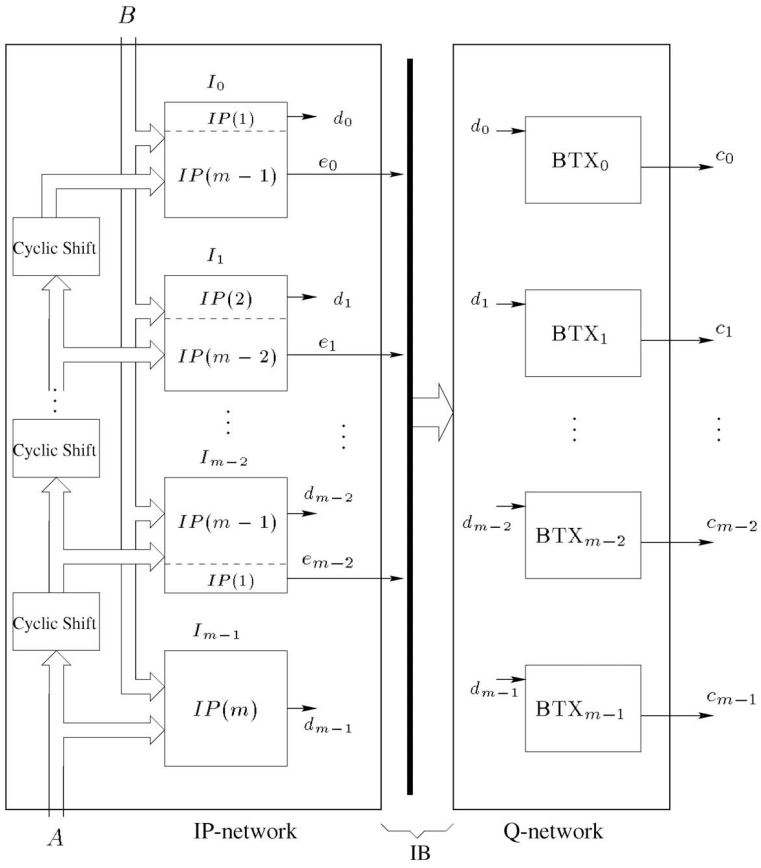


Figure 3.4: Architecture of the LCBP multiplier over $\mathbb{F}_{2^m}$ [18]

using cyclic shift and inner product units. The outputs of the inner product units are then split into two groups, d_i the bits of the product that do not need to be reduced, and e_i the bits of the product for which reduction is governed by $R(x)$. Then, IB selectively routes e_i bits to the appropriate BTX blocks in accordance with $R(x)$, effectively computing the reduction. The Q-network uses the BTX blocks (which are binary XOR trees) to determine the value of each bit of the output vector.

The Digit-Serial Approach

Finally, the digit-serial approach is one in which a digit width g is chosen and the result is calculated g bits at a time. This is done by dividing the multiplicand $B(x)$ into s portions

of width g, where $s = \lceil m/g \rceil$, and multiplying each portion by the multiplier $A(x)$, accumulating and shifting the partial result by g-bits. The details of this approach are shown in Algorithm 7, where $B_k(x)$ refers to a g-bit portion of $B(x)$.

As can be seen in Algorithm 7, in order for the digital-serial approach to be efficient the ability to perform the multiplication of an m-bit $\mathbb{F}_{2^m}$ element by a g-bit $\mathbb{F}_{2^m}$ element in a single clock cycle is a necessity. Since the structure of the bit-parallel multiplier presented in [18] is very regular and predictable, one of the inputs can be reduced to g-bits, while the other remains at m-bits, resulting in a multiplier that scales nicely in terms of size and clock frequency. By utilizing this approach a digit-width variable polynomial basis multiplier can be realized that can be set to produce a result in 1 to m clock cycles (given there are enough FPGA resources for the faster multipliers and that the multiplier meets the system's timing requirements).

Algorithm 7 Polynomial Basis Digit-Serial Multiplication [14]

INPUT: $A(x)$, $B(x)$ and $R(x)$
OUTPUT: $Z(x) = A(x) \times B(x) \bmod R(x)$
  $Z(x) \Leftarrow B_{s-1}(x)A(x) \bmod R(x)$
  for $i = s - 2$ downto 0 **do**
    $Z(x) \Leftarrow x^g Z(x)$
    $Z(x) \Leftarrow B_k(x)A(x) + Z(x) \bmod R(x)$
  end for

3.6.2 Polynomial Basis Squaring

It is fairly easy to prove that squaring a polynomial with binary coefficients is equivalent to shifting the coefficients to introduce zero coefficients between each of the already present coefficients. If the polynomial were to be viewed as a binary string, this would simply mean making the string twice as long by introducing zeros between the bit values.

$$A(x) = a_{m-1}x^{m-1} + a_{m-2}x^{m-2} + \ldots + a_1 x + a_0, \text{ where } a_i \in [0, 1] \tag{3.1}$$

$$A(x)^2 = a_{m-1}x^{2m-2} + a_{m-2}x^{2m-4} + \ldots + a_1 x^2 + a_0 \tag{3.2}$$

Knowing that squaring can be accomplished in hardware by routing signals, the next step is to simplify reduction. First, it must be observed that the lower half of the squared polynomial does not need to be reduced. Second, the reduction of the upper half of the polynomial can be simplified by factoring out a term of x^{m+1}.

$$(x^{m+1})(a_{m-1}x^{m-3} + a_{m-2}x^{m-5} + \ldots + a_{(m+3)/2}x^2 + a_{(m+1)/2}) \tag{3.3}$$

Also it should be noted that since reduction is being performed modulo the reduction polynomial $R(x) = x^m + x^d + \ldots + 1$ it holds true that $x^m \equiv x^d + \ldots + 1 \bmod R(x)$. Therefore, the reduction of the upper half of $A(x)^2$ can be computed as a multiplication by the lower coefficients of $R(x)$ shifted once to the left.

$$(x^{d+1} + \ldots + x)(a_{m-1}x^{m-3} + a_{m-2}x^{m-5} + \ldots + a_{(m+3)/2}x^2 + a_{(m+1)/2}) \tag{3.4}$$

Given that the architecture for a bit-parallel multiplier with a bit width variable input is available, this multiplication can be computed very efficiently since the g-bit input will be a sparse constant, as the reduction polynomials for the five recommended NIST fields are either pentanomials or trinomials, and the m-bit input will be half zeros. The final step in computing a square of a finite field element is simply to XOR the lower m bits of the expanded binary string with the output of the multiplier.

3.6.3 System Architecture

Two approaches were taken for the polynomial basis style scalar multiplier unit architecture. The first is a very simple blocking architecture in which each instruction sent to the hardware units blocks execution in the PPC Core until it is completely finished. This results in a simple resource efficient solution. However, this solution is completely serial and does not take advantage of the parallel nature of the FPGA. The second approach is a non-blocking architecture in which instructions sent to the hardware units start a hardware process and return immediately. The results of these instructions are not committed to the FPGA's registers until either a special instruction is called or another instruction of

the same type is sent to hardware. This approach allows for instructions to be pipelined and can reduce the cost of executing multi-cycle instructions. The downside to this approach is that it is the responsibility of the coder to reorder instructions in order to take advantage of pipelining, while taking care to avoid hazards. Also, in certain cases extra instruction calls must be made to be sure that an execution unit has completed and registered its result. The primary benefit of the non-blocking approach is that it may allow the polynomial basis approach to be comparable, and possibly even better, than the other two approaches as the a highly time consuming operation, the Frobenius map, can be computed in parallel with point additions.

3.7 Normal Basis Approach

When referring to elements of a binary Galois field being represented with regards to a normal basis, the elements are represented by the basis $\left\{\theta, \theta^2, \theta^{2^2}, \ldots, \theta^{2^{m-1}}\right\}$, where $\theta \in \mathbb{F}_{2^m}$ is chosen such that the basis is a set of linearly independent binary vectors. The use of a normal basis representation is an attractive approach as it reduces the computational cost of element squaring to virtually nothing when implemented in hardware. The downside to this approach is that multiplication is considerably more complex than compared to multiplication with elements represented in polynomial basis. However, there are hardware architectures available for fairly efficient normal basis squaring, such as the one presented in [19]. The other downside to a full normal basis approach is interoperability. Polynomial basis representation is perhaps the most commonly used representation of finite field elements and, therefore, when using a strict normal basis approach a method should be provided to convert back to polynomial basis for interoperability reasons.

3.7.1 Normal Basis Multiplication

The process of multiplying two elements represented in normal basis is much more complex and much less intuitive than with elements represented in polynomial basis. A primary

reason for this is that reduction is performed in a different manner than with polynomial basis representation. A simple way to view multiplication in a Galois field is that it is the process of multiplying two polynomials with binary coefficients followed by reduction of the product terms that do not fit the basis vectors. When the polynomials representing two normal basis vectors are multiplied, terms result that do not fit the basis vectors (i.e. θ^k where k is not a power of 2). These terms must be converted back into the normal basis representation and the coefficients associated with these terms must be redistributed likewise.

This redistribution is most often accomplished by utilizing a *multiplication matrix* $\boldsymbol{M}$ which defines how coefficients are to be redistributed. A *multiplication matrix* is defined based on θ and remains constant for all multiplications in a given field. Therefore, normal basis multiplication is most compactly defined as:

$$c_l = \underline{a}^{(l)} \cdot \mathbf{M} \cdot \underline{b}^{(l)^T} \tag{3.5}$$

where $\underline{a}^{(l)}$ denotes the l-bit left cyclic shift of the binary string representing element a.

Another way that normal basis multiplication is commonly defined is:

$$c_l = F(\underline{a}^{(l)}, \underline{b}^{(l)}) \tag{3.6}$$

where $F(\underline{u}, \underline{v}) = \sum_{k=1}^{p-2} u_{F(k+1)} v_{F(p-k)}$ and $F(x)$ is a function that is defined using similar methods which were used to create $\boldsymbol{M}$. In both cases the equations accomplish the same task and are evidently slower than polynomial basis multiplication as the computation of each bit requires a multiplication with a non-sparse matrix.

As mentioned earlier, efficient hardware architectures for computing normal basis multiplication do exist. Figure 3.5 shows the architecture for the digit-width variable multiplier presented in [19]. This is an efficient Gaussian Normal Basis (GNB) multiplier that can produce a product in one to m clock cycles, with the resource utilization of the multiplier increasing as the digit width g is increased.

The normal basis multiplier that was used in this thesis was implemented and studied by Glenn Ramsey in his work in [10]. Mr. Ramsey has been gracious enough to allow

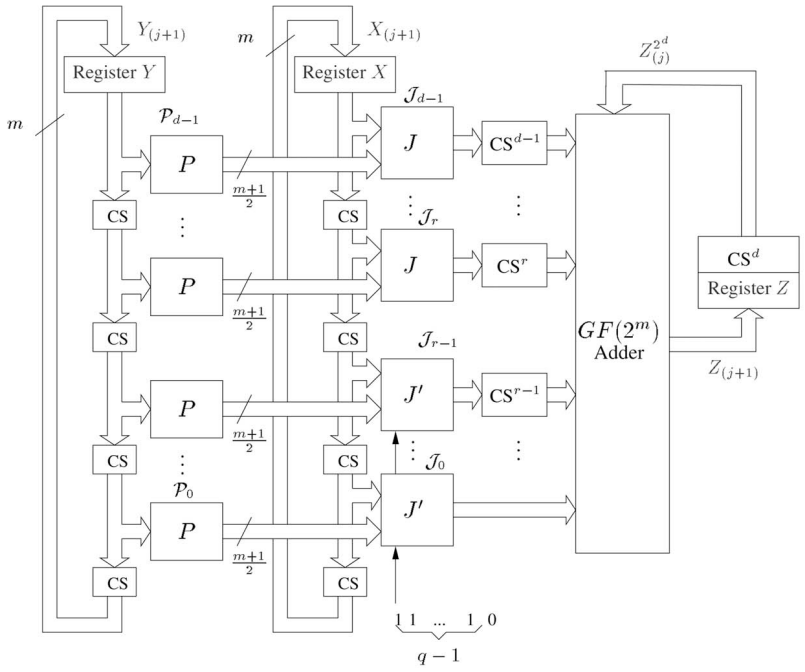


Figure 3.5: Digit-level GNB multiplier with parallel output [19]

the usage of his multiplier implementation for the work done in this thesis. Additional information on this multiplier and its performance can be found in [10] and [19].

3.7.2 Normal Basis Squaring

The primary benefit of utilizing a normal basis is the efficiency of the squaring operation. When an element is multiplied by itself in normal basis all of the terms that needed to be converted and redistributed in standard multiplication cancel with one another, therefore, eliminating the need for the multiplication matrix M. The only conversion that is required is that of the topmost bit as it becomes $c_{m-1}\theta^{2^m}$, and this is a simple conversion as it is known by Euler's theorem that $\theta^{2^m} \equiv \theta$. Therefore, squaring in normal basis is a cyclic shift, which can be implemented as routing in hardware.

In the case of Koblitz curves and RTNAF$_w$ the efficiency of normal basis squaring is even further exemplified as terms will need to be consecutively squared up to $m-1$ times.

Since up to 32-bit operands can be passed as operands to UDIs, a single instruction can square a normal basis finite field element up to 2^{32} times in a single clock cycle.

3.7.3 Finite Field Basis Conversion

Performing a basis conversion is the process of converting basis vectors from one basis representation to another. Basis conversion can simply be viewed as a mapping of one set of basis vectors to another and can be computed by a matrix multiplication. In order to convert an element to a different basis, the vector representing the element is multiplied by a matrix known as a conversion matrix which appropriately redistributes and combines the coefficient's of the given element. The process of creating the conversion matrices for the five NIST recommended curves is outlined in [17].

In the case of fields of the form $\mathbb{F}_{2^m}$ the conversion matrices have binary elements which simplifies the process of multiplying a vector by a matrix. When the vector and matrix are known to have binary elements this operation simply consists of a bitwise ANDing m vectors of length m and performing m parity checks on the resulting vectors. This results in simple hardware architectures that are resource efficient.

A digit width variable conversion unit was created for use in this thesis, where the concept of a digit width variable unit is the same as what was presented in the context of the Galois field multipliers mentioned previously. In the case of a full digit width setting a fully combinational unit was implemented where all m bits of the output vector were calculated at once. Due to the fact that the basis conversion matrices are not sparse, this did not always result in a compact solution, especially for the larger field sizes. For the smaller digit widths, which required a greater number of clock cycles in order to produce a result, the units were realized by storing the conversion matrix in ROM and calculating d output bits at a time, where d was the unit's digit width. This approach allowed for a conversion unit that could vary in runtime and resource requirements depending on the requirements of the system.

3.7.4 System Architecture

As was the case with the polynomial basis approach, both a blocking and non-blocking system approach were investigated. However, in the case of this architecture the development of the non-blocking approach was considerably less significant as the multiplication unit was the only major operation to benefit from its implementation.

3.8 Mixed Basis Approach

The enticement of a mixed basis approach is that a system can be created which benefits from computationally negligible squaring, when utilizing a normal basis, and relatively fast multiplication, when using a polynomial basis. The drawback to this approach is that elements will need to be converted between the two bases, and this must be done quickly and with as little additional resource utilization as possible.

3.8.1 Finite Field Multiplication

Finite field multiplication will be performed using the polynomial basis multiplier architecture described in Section 4.5.1.

3.8.2 Finite Field Squaring

The squaring of a finite field element will be computed in one of two ways. The first will be to square an element represented in polynomial basis as described in Section 4.5.2. This will be done when fewer than three consecutive squarings are required and it is not profitable to convert from one basis to another and back again. In the case that more than three consecutive squarings are required, such as when applying the Frobenius map, elements will be represented in normal basis and will be squared as described in Section 4.6.2. The reason for this distinction is that although squaring in normal basis is essentially a computationally free operation, basis conversion is not. The polynomial basis squaring

unit presented in Section 4.5.2 is of a negligible size and can perform a single squaring per clock cycle. Therefore, given that polynomial basis is the preferred basis, due to its ubiquity and that it is the one that the final result will ultimately be presented in, in certain situations it is more computationally feasible to square polynomial basis elements as opposed to normal basis elements.

3.8.3 Finite Field Basis Conversion

Basis conversion will be completed as described in Section 4.6.3. In the case of this approach, however, two basis conversion units will be required so that elements can be converted from polynomial basis to normal basis as well as from normal basis to polynomial basis.

3.8.4 System Architecture

As was the case with the previous two approaches a blocking and non-blocking architecture will be created and compared for this approach. As was the case with the normal basis approach, it is expected that non-blocking approach is less crucial than in the case of the polynomial basis approach as it will only offer speedup in the case of multipliers with smaller digit widths.

3.9 Double-and-Add Approach

Finally, as a means of internal comparison, the double-and-add approach will be implemented to compute scalar multiplication on Koblitz curves. Unlike the utilization of RTNAF$_w$ and the Frobenius map, the double-and-add method is not exclusive to Koblitz curves and offers no additional benefit when applied to Koblitz curves as compared to generic elliptic curves. The algorithm for double-and-add is shown in Algorithm 8.

Algorithm 8 Double-and-Add Method for Computing Scalar Multiplication on Elliptic Curves [4]

INPUT: $k = (k_{l-1}, \ldots, k_1, k_0)_2$, a positive integer less than $r/2$
P, a point on a given elliptic curve
OUTPUT: The point kP
$Q \Leftarrow \mathcal{O}$
for i from $l-1$ downto 0 **do**
$Q = 2Q$
if $(k_i == 1)$ **then**
$Q = Q + P$
end if
end for

Chapter 4

Implementation Results

The implementation results for this thesis are presented in two sections. First, the resource utilization and runtimes for the individual hardware units are presented and discussed. This section gives a low level view of how each major hardware component performed individually and compares the normal basis and polynomial basis units to one another. Second, the performance of the three system approaches: the polynomial basis approach, normal basis approach and mixed basis approach, are presented and discussed. This section compares the three different approaches to one another and shows how the systems scale in terms of resource utilization and runtime for different parameter settings in the $\mathbb{F}_{2^{163}}$ case and shows which system was the fastest for each basis given the resources available on the FX60.

Relative efficiency is measured and compared based upon the execution time and total slice utilization for a unit. The relative efficiency value used to compare units was computed as $(\textit{execution time} \times \textit{total slice utilization})^{-1}$. With this metric units that have the quickest execution time while utilizing the fewest resources were ranked as most efficient.

4.1 Hardware Unit Results

There are four primary hardware unit types used in the a scalar multiplication system: the finite field multipliers, the finite field squaring units, the finite field inversion units and the basis conversion units. For each unit type the relative efficiency of the polynomial and normal basis variants are compared over each field for a variety of runtimes, keeping

in mind that the units have different execution times depending on either their parameter settings or the specific operation they are used for. The units used for relative efficiency in this section are (milliseconds $\times$ *total number of slices*)$^{-1}$. It should also be noted that in the tables provided below the "Eff" column contains data on the unit's relative efficiency and the "% Slices," "% FFs," and "% LUTs" columns refer to the percentage of the FX60's resources that have been used to implement the unit.

4.1.1 Finite Field Multipliers

The efficiency of the finite field multiplication unit is arguably one of the most important factors in the performance of a scalar multiplication unit as finite field multiplication is the most frequent nontrivial operation performed when calculating a scalar multiple on a Koblitz curve. In the case of this thesis two multiplier architectures were utilized. A polynomial basis multiplier was used which was created as described in Section 4.5.1 and a normal basis multiplier was used as described in Section 4.6.1. The normal basis multiplier was implemented by Glenn Ramsey Jr. as part of his work in [10] and permission was granted for it to be used as part of the work done in this thesis.

Given the significance of multiplication in the scalar multiplication algorithm, the efficiency comparison of these two multipliers will offer insight into how the polynomial basis approach will perform in comparison to the normal basis approach. For reasons discussed in Section 4.5 in order for the polynomial basis approach to compete with the normal basis approach, the polynomial basis multiplier must be significantly more efficient. Judging by the architectural differences between the polynomial and normal basis multipliers it appears that the polynomial basis multiplier is inherently more resource efficient, but the question of concern is whether the difference in efficiency between the two multipliers is significant enough to give the polynomial basis approach the upper hand.

Table 4.1: $\mathbb{F}_{2^{163}}$ Polynomial Basis Multiplier Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	163	1.33	462	(1%)	516	(1%)	420	(0%)
2	82	2.69	454	(1%)	510	(1%)	746	(1%)
4	41	5.21	468	(1%)	515	(1%)	871	(1%)
8	21	6.40	744	(2%)	560	(1%)	1367	(2%)
17	10	5.81	1721	(6%)	562	(1%)	3296	(6%)
21	8	5.97	2095	(8%)	566	(1%)	4027	(7%)
41	4	6.70	3730	(14%)	640	(1%)	7104	(14%)
82	2	8.06	6215	(24%)	1011	(1%)	11805	(23%)
163	1	8.47	11809	(46%)	1992	(3%)	22407	(44%)

Table 4.2: $\mathbb{F}_{2^{163}}$ Normal Basis Multiplier Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	163	1.11	554	(2%)	694	(1%)	1036	(2%)
2	82	1.79	681	(2%)	669	(1%)	1294	(2%)
4	41	3.10	786	(3%)	665	(1%)	1440	(2%)
8	21	3.94	1210	(4%)	697	(1%)	2297	(4%)
17	10	4.14	2413	(9%)	958	(1%)	4663	(9%)
21	8	4.45	2810	(11%)	1027	(2%)	5422	(10%)
41	4	4.45	5617	(22%)	1223	(2%)	10850	(21%)
82	2	4.68	10685	(42%)	1815	(3%)	20677	(40%)
163	1	5.92	16895	(66%)	3943	(7%)	33057	(65%)

Tables 4.1 and 4.2 show the resource utilization and relative efficiency for the polynomial basis and normal basis multipliers, respectively. The digit widths chosen in these tables covers the span of the field size and have been chosen to maximize resource efficiency. The number of clock cycles that are required for a digit width variable multiplier to produce a product is calculated by computing $\lceil m/g \rceil$ where g is the multiplier digit width and m is the field size. In order to produce the most efficient multiplier g should be chosen such that m/g is as close as possible to $\lceil m/g \rceil$. In this way the final partial calculation produces as close to g bits of the product as possible and resources are used more efficiently.

There are several significant conclusions to gather from Table 4.1 and Table 4.2. First of all, in the case of the normal basis multiplier for each increase in digit width there is also an increase in efficiency, with the largest multiplier being the most efficient. This is

generally the case with the polynomial basis multiplier, however, for $g = 21$ there is an unusual spike in efficiency. Still, as was the case with the normal basis multiplier, the largest polynomial basis multiplier is the most efficient. It is also important to notice that while both multipliers show an overall growth in efficiency as the digit width is increased, this growth is not linear. The largest growth is seen in the lower digit widths, while in the case of the larger digit width multipliers the multiplication speed is reduced by less than ten cycles per digit width increase yet the resource utilization still nearly doubles.

As far as comparing the efficiency of the two multipliers the polynomial basis multiplier is consistently more efficient than the normal basis multiplier, as was to be expected by studying their respective architectures. The disparity in the efficiency of the two multipliers shows an overall growth as the digit width is increased, with the $g = 2$ offering the largest growth. Overall, it is still unclear as to whether or not the disparity in efficiency between the two multipliers is great enough for the polynomial basis system approach to gain the upper hand. However, this data does show that the polynomial basis system approach should at least be a strong contender to the normal basis system approach.

Table 4.3: $\mathbb{F}_{2^{233}}$ Polynomial Basis Multiplier Resource Utilization

Digit Width	**Cycles**	**Eff**	**Slices**	**% Slices**	**FFs**	**% FFs**	**LUTs**	**% LUTs**
1	233	0.74	580	(2%)	725	(1%)	472	(0%)
2	117	1.37	624	(2%)	723	(1%)	1018	(2%)
4	59	2.54	668	(2%)	723	(1%)	1258	(2%)
8	30	3.21	1039	(4%)	735	(1%)	1881	(3%)
15	16	2.95	2118	(8%)	748	(1%)	4070	(8%)
30	8	3.27	3804	(15%)	777	(1%)	7223	(14%)
59	4	3.41	7328	(28%)	921	(1%)	13953	(27%)
117	2	4.00	12495	(49%)	3195	(6%)	23701	(46%)
233	1	4.18	23912	(94%)	6962	(13%)	45373	(89%)

Table 4.4: $\mathbb{F}_{2^{233}}$ Normal Basis Multiplier Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	233	0.60	721	(2%)	949	(1%)	1351	(2%)
2	117	1.00	857	(3%)	985	(1%)	1607	(3%)
4	59	1.47	1155	(4%)	946	(1%)	2197	(4%)
8	30	1.24	2697	(10%)	948	(1%)	5203	(10%)
15	16	1.15	5417	(21%)	1726	(3%)	10600	(20%)
30	8	1.24	10063	(39%)	1585	(3%)	19824	(39%)
59	4	1.21	20723	(81%)	3284	(6%)	41006	(81%)
117	2	1.71	29282	(115%)	1031	(2%)	57983	(114%)
233	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Tables 4.3 and 4.4 show the resource utilization and relative efficiencies for the polynomial and normal basis multipliers for the NIST recommended field of $\mathbb{F}_{2^{233}}$. The data presented for these multipliers is similar to the data presented for the $\mathbb{F}_{2^{163}}$ multipliers. Again there is an overall increase in resource utilization and efficiency as the digit width is increased, and again the gap in efficiency between the polynomial basis and normal basis multiplier increases with the digit width. In the case of the full digit width multiplier in Table 4.4 the efficiency and resource utilization data have been denoted as "N/A." The reason for this is that the Xilinx synthesis tool ran out of memory when attempting to synthesize the given system. This becomes more prevalent in the larger field sizes.

Table 4.5: $\mathbb{F}_{2^{283}}$ Polynomial Basis Multiplier Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	283	0.48	732	(2%)	883	(1%)	589	(1%)
2	142	0.88	802	(3%)	904	(1%)	1302	(2%)
4	71	1.89	774	(3%)	880	(1%)	1446	(2%)
8	36	2.14	1300	(5%)	871	(1%)	2384	(4%)
18	16	2.21	2826	(11%)	933	(1%)	5449	(10%)
36	8	2.42	5160	(20%)	1217	(2%)	9874	(19%)
71	4	2.40	10420	(41%)	1273	(2%)	19786	(39%)
142	2	2.71	18466	(73%)	3102	(6%)	35043	(69%)
283	1	2.83	35395	(140%)	6878	(13%)	67169	(132%)

Table 4.6: $\mathbb{F}_{2^{283}}$ Normal Basis Multiplier Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	283	0.37	956	(3%)	1151	(2%)	1780	(3%)
2	142	0.56	1254	(4%)	1192	(2%)	2345	(4%)
4	71	0.81	1731	(6%)	1145	(2%)	3278	(6%)
8	36	1.08	2571	(10%)	1403	(2%)	4923	(9%)
18	16	1.21	5159	(20%)	2181	(4%)	9976	(19%)
36	8	1.24	10101	(39%)	2237	(4%)	19520	(38%)
71	4	1.31	18975	(75%)	4294	(8%)	36846	(72%)
142	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
283	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 4.7: $\mathbb{F}_{2^{409}}$ Polynomial Basis Multiplier Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	409	0.23	1056	(4%)	1254	(2%)	882	(1%)
2	205	0.47	1038	(4%)	1270	(2%)	1620	(3%)
4	103	0.88	1107	(4%)	1269	(2%)	2065	(4%)
8	52	1.05	1831	(7%)	1264	(2%)	3341	(6%)
16	26	1.11	3457	(13%)	1473	(2%)	6582	(13%)
30	14	1.12	6371	(25%)	1364	(2%)	12084	(23%)
52	8	1.16	10784	(42%)	2207	(4%)	20476	(40%)
103	4	1.18	21201	(83%)	1536	(3%)	40437	(79%)
205	2	1.31	38165	(150%)	7750	(15%)	72351	(143%)
409	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 4.8: $\mathbb{F}_{2^{409}}$ Normal Basis Multiplier Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	409	0.18	1389	(5%)	1717	(3%)	2592	(5%)
2	205	0.27	1797	(7%)	1654	(3%)	3414	(6%)
4	103	0.48	2037	(8%)	1654	(3%)	3820	(7%)
8	52	0.59	3239	(12%)	1754	(3%)	6269	(12%)
16	26	0.66	5817	(23%)	2305	(4%)	11291	(22%)
30	14	0.77	9180	(36%)	3320	(6%)	17788	(35%)
52	8	N/A	N/A	N/A	N/A	N/A	N/A	N/A
103	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A
205	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
409	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 4.9: $\mathbb{F}_{2^{571}}$ Polynomial Basis Multiplier Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	571	0.12	1499	(5%)	1752	(3%)	1258	(2%)
2	286	0.24	1481	(5%)	1759	(3%)	2383	(4%)
4	143	0.43	1630	(6%)	1753	(3%)	3068	(6%)
8	72	0.54	2590	(10%)	1776	(3%)	4734	(9%)
16	36	0.57	4835	(19%)	2063	(4%)	9213	(18%)
29	20	0.56	8986	(35%)	1910	(3%)	17031	(33%)
44	13	0.61	12592	(49%)	2574	(5%)	23922	(47%)
72	8	0.62	20192	(79%)	3750	(7%)	38363	(75%)
143	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A
286	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
571	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Table 4.10: $\mathbb{F}_{2^{571}}$ Normal Basis Multiplier Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	571	0.07	2270	(8%)	2314	(4%)	4196	(8%)
2	286	0.12	2992	(11%)	2303	(4%)	5511	(10%)
4	143	0.16	4394	(17%)	2605	(5%)	8313	(16%)
8	72	0.19	7271	(28%)	3730	(7%)	13702	(27%)
16	36	0.19	14468	(57%)	3702	(7%)	27297	(53%)
29	20	0.20	25611	(101%)	2295	(4%)	48401	(95%)
44	13	0.20	37221	(147%)	5193	(10%)	70891	(140%)
72	8	N/A	N/A	N/A	N/A	N/A	N/A	N/A
143	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A
286	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
571	1	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Overall the data presented for the $\mathbb{F}_{2^{233}}$ multipliers (Tables 4.3 and 4.4), $\mathbb{F}_{2^{283}}$ multipliers (Tables 4.5 and 4.6), $\mathbb{F}_{2^{409}}$ multipliers (Tables 4.7 and 4.8), and $\mathbb{F}_{2^{571}}$ multipliers (Tables 4.9 and 4.10) offers little additional information, as it could have been extrapolated from the data presented on the $\mathbb{F}_{2^{163}}$ multipliers. These tables are shown to give an idea of the resources required for the larger fields multipliers and to show that in the case of the larger field implementations very fast multipliers may not be a valid option on the FX60. As mentioned earlier, the efficiency disparity between the polynomial and normal basis multipliers grows as the digit widths are increased and the FX60 may not being able to handle ideal parameter settings for each of the system approaches. This may result in a situation where a system that is most efficient for $\mathbb{F}_{2^{163}}$ is not the most efficient choice in the case of the larger fields due to the fact that the disparity between the multiplier efficiencies was not significant enough.

4.1.2 Finite Field Squaring Unit

Low resource utilization squaring units that require a single clock cycle to square an element can be implemented for both the polynomial basis (see Section 4.5.2 for details) and the normal basis (see Section 4.6.2 for details). In the case of this thesis not only was it important to create a squaring unit that could compute product in a single clock cycle, but it

was also necessary to create a unit that could efficiently perform consecutive squaring on a given element. In the case of the polynomial basis squaring unit this resulted in the addition of a state machine and additional memory to produce a unit that requires n clock cycles to consecutively square an element n times. In the case of the normal basis squaring unit this resulted in the addition of considerable routing to produce a unit that requires a single clock cycle regardless of the number of consecutive squarings requested. Tables 4.11 and 4.12 show the resource utilization for the squaring units used in this thesis.

Table 4.11: Polynomial Basis Squaring Unit Resource Utilization

Field Size (m)	**Slices**	**% Slices**	**FFs**	**% FFs**	**LUTs**	**% LUTs**
163	225	(0%)	191	(0%)	437	(0%)
233	254	(1%)	260	(0%)	498	(0%)
283	382	(1%)	326	(0%)	741	(1%)
409	450	(1%)	466	(0%)	892	(1%)
571	759	(3%)	671	(1%)	1473	(2%)

Table 4.12: Normal Basis Squaring Unit Resource Utilization

Field Size (m)	**Slices**	**% Slices**	**FFs**	**% FFs**	**LUTs**	**% LUTs**
163	750	(2%)	0	(0%)	1304	(2%)
233	1072	(4%)	0	(0%)	1864	(3%)
283	1465	(5%)	0	(0%)	2547	(5%)
409	2117	(8%)	0	(0%)	3681	(7%)
571	3283	(12%)	0	(0%)	5710	(11%)

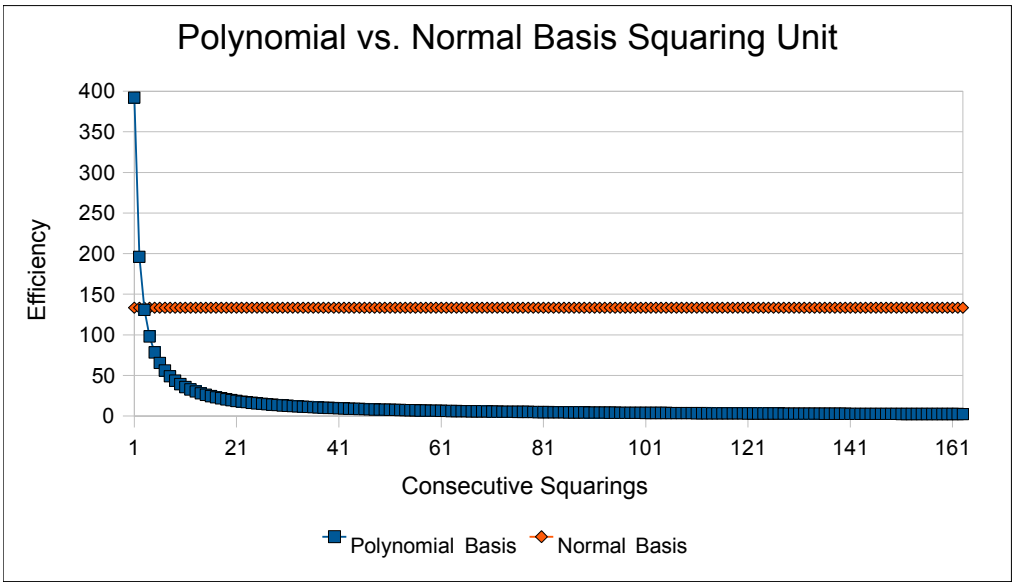


Figure 4.1: Polynomial vs. Normal Basis Squaring Unit Efficiency $\mathbb{F}_{2^{163}}$

Figures 4.1, 4.2, 4.3, 4.4 and 4.5 have been provided to show the variation in efficiency of each squaring unit as the number of consecutive squarings is varied from a single squaring to m consecutive squarings. This data gives a better idea of how the normal basis and polynomial basis squaring units compare in the context of scalar multiplication on Koblitz curves. In the case of a single squaring the polynomial basis unit proves to be more efficient as it requires fewer resources. However, as the number of squarings is increased the runtime of the polynomial basis unit increases while the runtime of the normal basis unit remains constant, which allows the normal basis unit to quickly take the lead in efficiency. This is particularly significant when performing scalar multiplier on Koblitz curves as elements often need to be consecutively squared a large number of times. However, it is important to keep in mind that these efficiency comparisons neglect the use of the non-blocking system approach mentioned in section 4.5.2. This approach should help alleviate the efficiency problem with the polynomial basis squaring unit. Of course these techniques will not be applied in the case of the blocking variation of the system approaches and the effects of the data shown in 4.1, 4.2, 4.3, 4.4 and 4.5 should be evident in the comparisons of these systems.

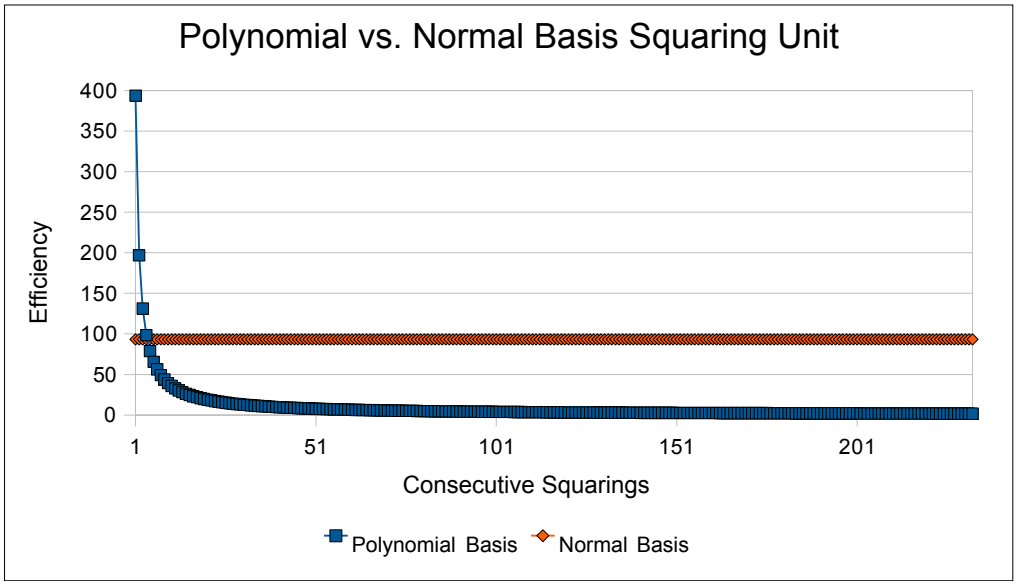


Figure 4.2: Polynomial vs. Normal Basis Squaring Unit Efficiency $\mathbb{F}_{2^{233}}$

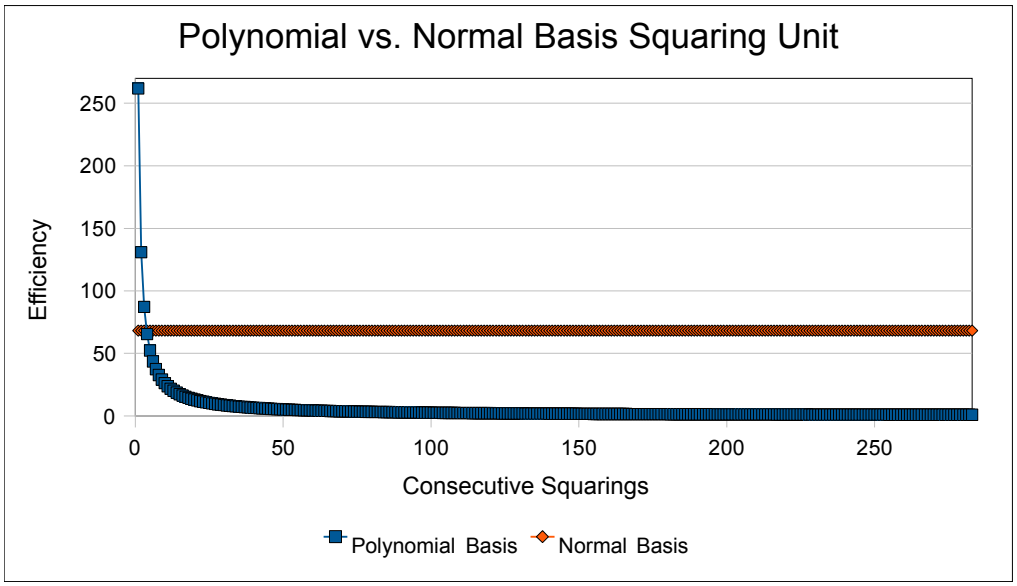


Figure 4.3: Polynomial vs. Normal Basis Squaring Unit Efficiency $\mathbb{F}_{2^{283}}$

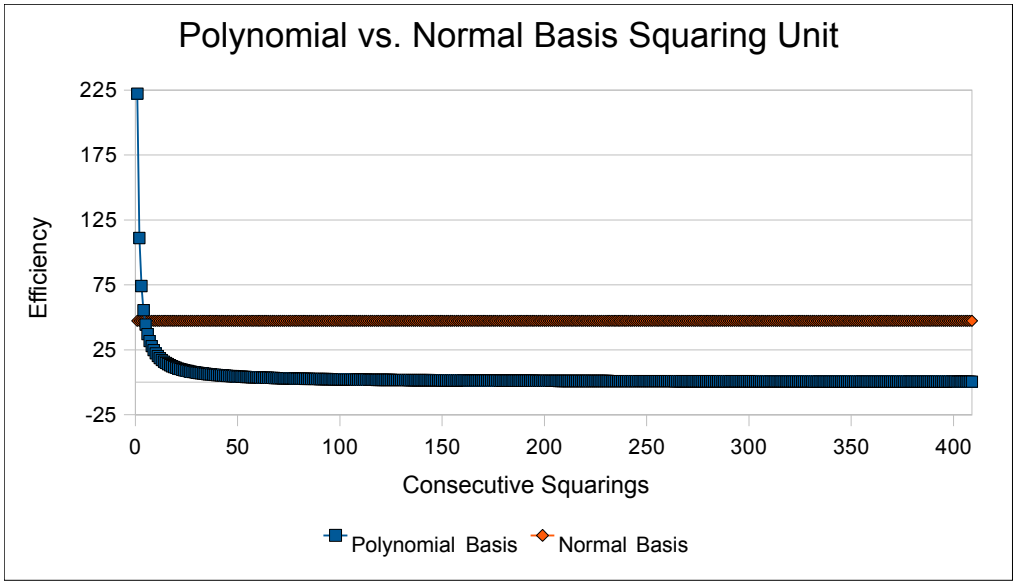


Figure 4.4: Polynomial vs. Normal Basis Squaring Unit Efficiency $\mathbb{F}_{2^{409}}$

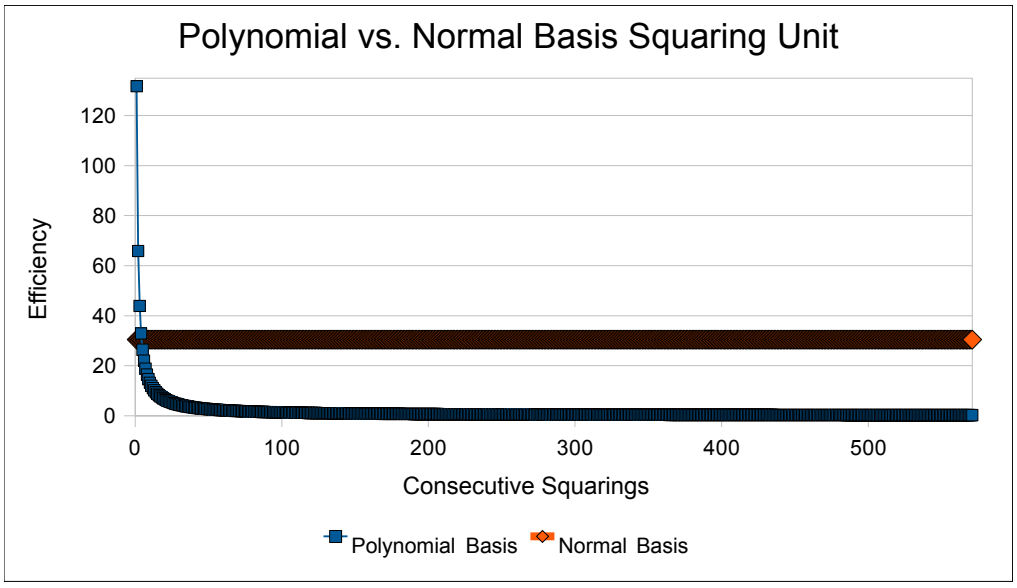


Figure 4.5: Polynomial vs. Normal Basis Squaring Unit Efficiency $\mathbb{F}_{2^{571}}$

4.1.3 Finite Field Inversion Unit

When computing a scalar multiple on a Koblitz curve the ability to calculate finite field inverses is a necessity. In the case of this thesis projective coordinates were used and thus inverse calculations were only required when converting an elliptic curve point from projective coordinates to affine. This resulted in a system where the time required to calculate an inverse was not of a particularly high priority as it only occurred once per scalar multiple. However, the inversion unit was still implemented in hardware as it did not require a significant amount of resources, due primarily to the reuse of the system's multiplier, and was assumed to be significantly faster than a software approach. As detailed in Section 4.3, Itoh and Tsujii's inversion algorithm was used to perform field inversions, which is an algorithm that has a predictable runtime for a given field. This runtime along with the resource utilization for the polynomial basis and normal basis units are shown in Tables 4.13 and 4.14 respectively. In these tables s represents the number of clock cycles required for a multiplication to be computed.

Table 4.13: Polynomial Basis Inversion Unit Resource Utilization

Field Size (m)	**Cycles**	**Slices**	**% Slices**	**FFs**	**% FFs**	**LUTs**	**% LUTs**
163	$9 \times s + 162$	866	(3%)	441	(0%)	1649	(3%)
233	$10 \times s + 232$	1120	(4%)	581	(1%)	2144	(4%)
283	$11 \times s + 282$	1394	(5%)	684	(1%)	2647	(5%)
409	$11 \times s + 408$	1848	(7%)	946	(1%)	3508	(6%)
571	$13 \times s + 570$	2684	(10%)	1277	(2%)	5076	(10%)

Table 4.14: Normal Basis Inversion Unit Resource Utilization

Field Size (m)	**Cycles**	**Slices**	**% Slices**	**FFs**	**% FFs**	**LUTs**	**% LUTs**
163	$9 \times s + 10$	1613	(6%)	408	(0%)	3045	(6%)
233	$10 \times s + 11$	2306	(9%)	586	(1%)	4355	(8%)
283	$11 \times s + 12$	2936	(11%)	704	(1%)	5573	(11%)
409	$11 \times s + 12$	4384	(17%)	1001	(1%)	8327	(16%)
571	$13 \times s + 14$	5721	(22%)	1399	(2%)	11006	(21%)

One comparison that immediately stands out is that the normal basis inversion unit is inherently faster than the polynomial basis inversion unit. This is due to the consecutive

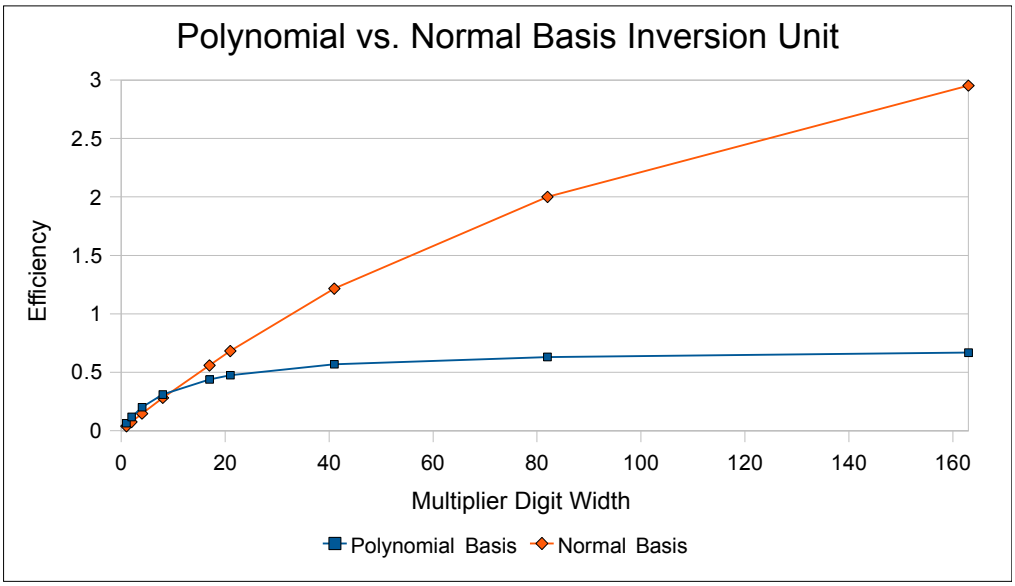


Figure 4.6: Polynomial vs. Normal Basis Inversion Unit Efficiency $\mathbb{F}_{2^{163}}$

squarings that are required in Itoh and Tsujii's inversion algorithm and the fact that this operation can be quickly and easily computed in normal basis representation. Unlike the case of the non-blocking polynomial basis system architecture, detailed in Section 4.5.3, no steps were taken to alleviate the single squaring per clock cycle inefficiency of the polynomial basis squaring unit. The reason for this is that inversion only occurs once per scalar multiplication and the cost in terms of development and resources was judged to not be worth the small gain in runtime.

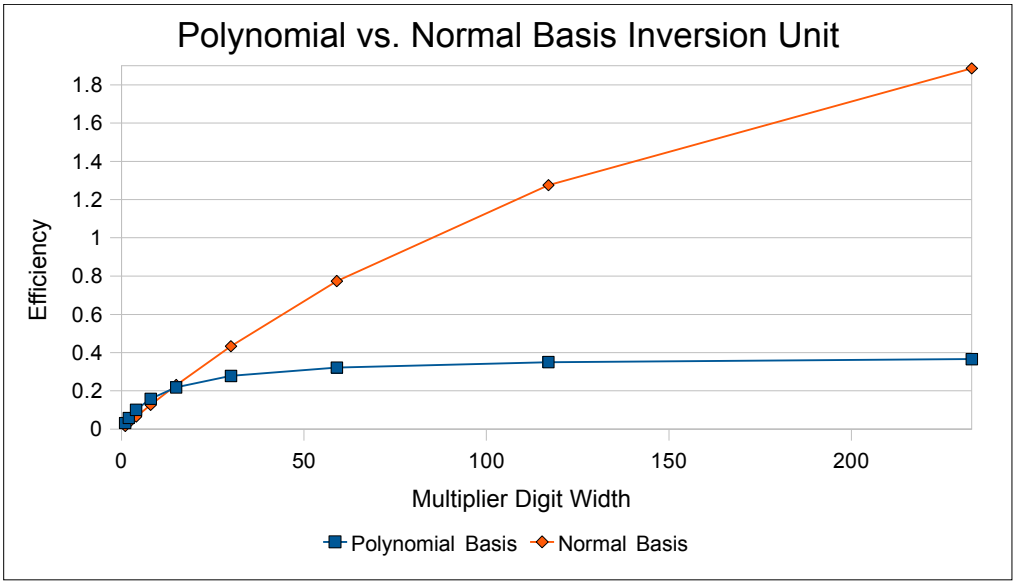


Figure 4.7: Polynomial vs. Normal Basis Inversion Unit Efficiency $\mathbb{F}_{2^{233}}$

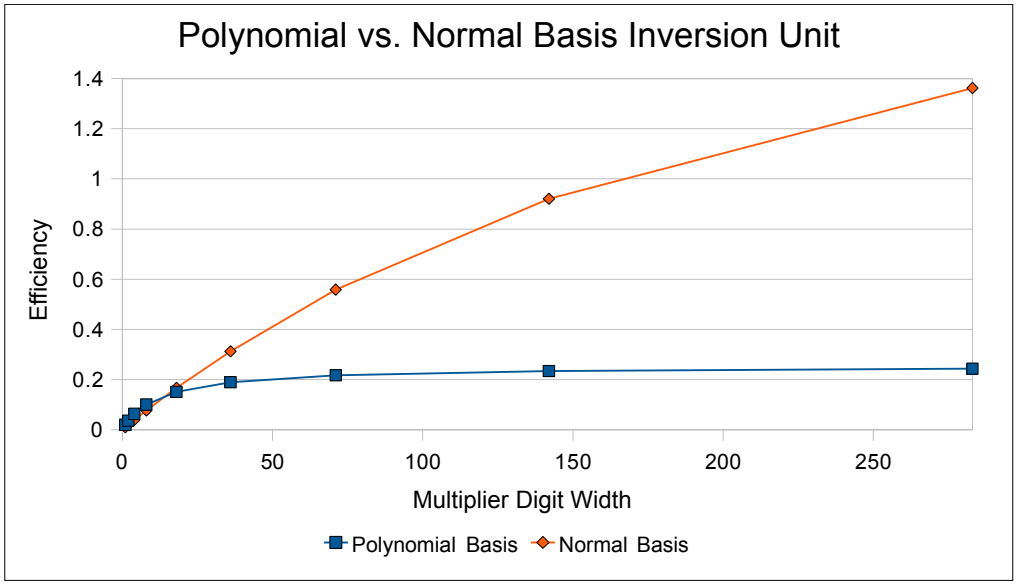


Figure 4.8: Polynomial vs. Normal Basis Inversion Unit Efficiency $\mathbb{F}_{2^{283}}$

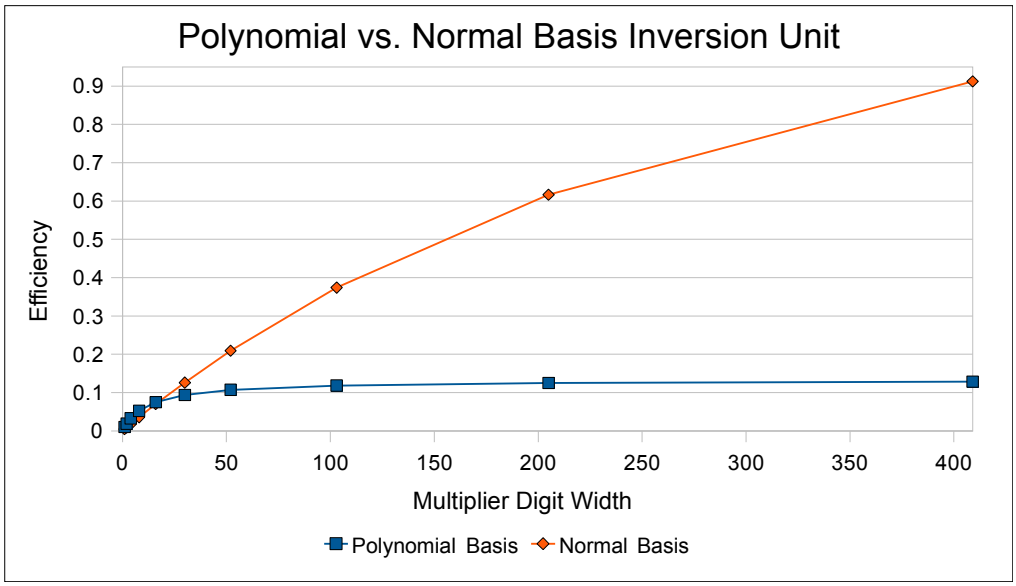


Figure 4.9: Polynomial vs. Normal Basis Inversion Unit Efficiency $\mathbb{F}_{2^{409}}$

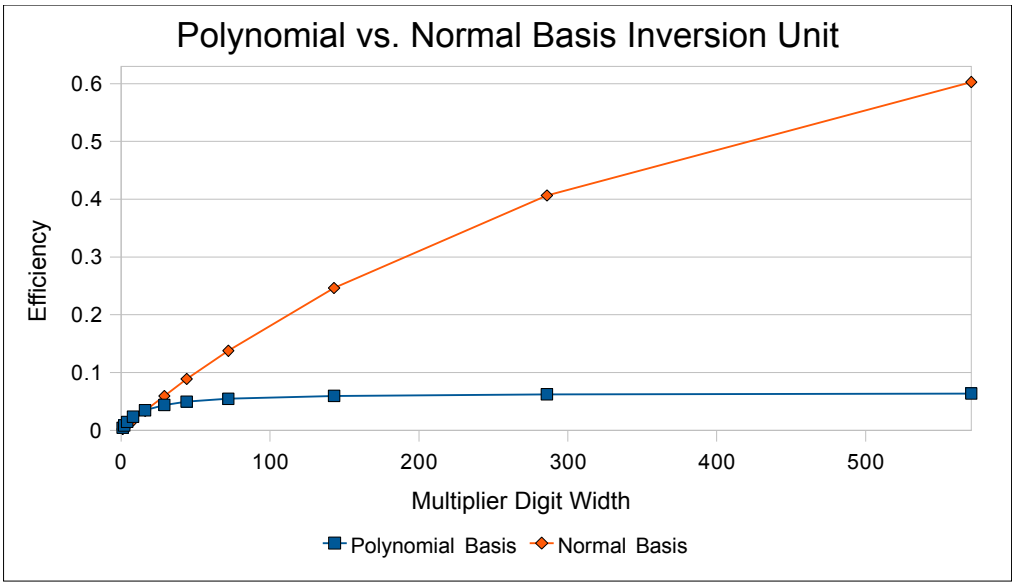


Figure 4.10: Polynomial vs. Normal Basis Inversion Unit Efficiency $\mathbb{F}_{2^{571}}$

Figures 4.6, 4.7, 4.8, 4.9 and 4.10 show how the efficiency of the polynomial and normal basis inversion units digress as the speed of the multiplier is increased. In each of these five cases a threshold is crossed at which the efficiency of the normal basis unit becomes greater than that of the polynomial basis. The reason for this is that in the case of a slower multiplier the additional clock cycles required for squaring in the polynomial basis are insignificant compared to time required to perform the necessary multiplications. However, as the multiplier's speed is increased the efficiency of normal basis squaring becomes more significant and the efficiency of the normal basis inversion unit grows at a greater rate than that of the polynomial basis unit. The results of this data contrast the data collected in relation to the multipliers where increasing the digit widths lead the polynomial basis unit gaining a greater margin in efficiency. However, given how infrequent the inversion operation is, this data may not have too significant of an impact on the scalar multiplication system approaches.

4.1.4 Basis Conversion Unit

In the case of the normal basis approach the ability to perform basis conversion is required as the final result must be converted back to the ubiquitous polynomial basis representation. The speed at which this operation completes is not critical in the case of the normal basis approach, as the operation is only performed once on the final result, however, in the case of the mixed basis approach multiple conversions are required from normal basis to polynomial basis and vice versa. In order to aid in flexibility for the situation of the mixed basis approach, as well as the normal basis approach, where resources might be better used for a faster multiplier, a digit width variable basis conversion unit was created.

Table 4.15: $\mathbb{F}_{2^{163}}$ Converter Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	163	3.28	187	(0%)	173	(0%)	338	(0%)
2	82	6.01	203	(0%)	172	(0%)	371	(0%)
4	41	8.05	303	(1%)	171	(0%)	557	(1%)
8	21	8.72	546	(2%)	170	(0%)	1004	(1%)
17	10	7.80	1282	(5%)	171	(0%)	2351	(4%)
21	8	4.33	2889	(11%)	272	(0%)	5435	(10%)
41	4	5.96	4197	(16%)	351	(0%)	7864	(15%)
82	2	0.88	56565	(223%)	13569	(26%)	109965	(217%)
163**(P2N)**	1	54.88	1822	(7%)	1	(0%)	3189	(6%)
163**(N2P)**	1	54.76	1826	(7%)	1	(0%)	3196	(6%)

Table 4.15 shows the resource utilization for the conversion units for $\mathbb{F}_{2^{163}}$. The reason that both the polynomial to normal basis and the normal to polynomial basis conversion unit statistics are shown in a single table is that in the cases for which the digit width is not equal to m the only difference between the two units is the conversion matrix which is stored in ROM and, therefore, does not affect resource utilization. In the case of the full digit width unit the system is fully combinational and the conversion direction is specified in the "Digit Width" column.

One important observation to be gathered from this data is that selecting a digit width less than m is not always a wise choice. The reason for this is that the full digit width unit is fully combinational while the partial digit width variants are not. In the case of the partial digit width configurations memory controllers are needed to supply a certain number of columns of the conversion matrix every clock cycle. The extra logic needed for these memory controllers eventually becomes greater than the resources required in the situation in which a fully combinational approach is taken.

Table 4.16: $\mathbb{F}_{2^{233}}$ Converter Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	233	1.63	263	(1%)	245	(0%)	479	(0%)
2	117	2.75	311	(1%)	244	(0%)	573	(1%)
4	59	3.69	459	(1%)	243	(0%)	847	(1%)
8	30	4.18	798	(3%)	242	(0%)	1471	(2%)
15	16	2.39	2618	(10%)	311	(0%)	4926	(9%)
30	8	2.79	4473	(17%)	461	(0%)	8345	(16%)
59	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A
117	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
233**(P2N)**	1	28.79	3474	(13%)	1	(0%)	6083	(12%)
233**(N2P)**	1	28.71	3483	(13%)	1	(0%)	6094	(12%)

Table 4.17: $\mathbb{F}_{2^{283}}$ Converter Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	283	1.10	322	(1%)	294	(0%)	584	(1%)
2	142	1.93	365	(1%)	295	(0%)	670	(1%)
4	71	2.15	655	(2%)	306	(0%)	1212	(2%)
8	36	2.57	1080	(4%)	302	(0%)	1993	(3%)
18	16	1.98	3156	(12%)	375	(0%)	5886	(11%)
36	8	0.21	59391	(234%)	10676	(21%)	116104	(229%)
71	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A
142	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
283**(P2N)**	1	19.86	5036	(19%)	1	(0%)	8796	(17%)
283**(N2P)**	1	19.81	5048	(19%)	1	(0%)	8825	(17%)

Table 4.18: $\mathbb{F}_{2^{409}}$ Converter Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	409	0.56	437	(1%)	421	(0%)	786	(1%)
2	205	0.93	527	(2%)	422	(0%)	965	(1%)
4	103	1.21	803	(3%)	420	(0%)	1478	(2%)
8	52	1.35	1421	(5%)	419	(0%)	2616	(5%)
16	26	1.19	3227	(12%)	519	(1%)	5974	(11%)
30	14	N/A	N/A	N/A	N/A	N/A	N/A	N/A
52	8	N/A	N/A	N/A	N/A	N/A	N/A	N/A
103	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A
205	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
409**(P2N)**	1	10.23	9778	(38%)	1	(0%)	17075	(33%)
409**(N2P)**	1	10.17	9837	(38%)	1	(0%)	17171	(33%)

Table 4.19: $\mathbb{F}_{2^{571}}$ Converter Resource Utilization

Digit Width	Cycles	Eff	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
1	571	0.29	601	(2%)	584	(1%)	1080	(2%)
2	286	0.52	669	(2%)	583	(1%)	1218	(2%)
4	143	0.68	1024	(4%)	583	(1%)	1876	(3%)
8	72	0.02	48966	(193%)	17986	(35%)	82711	(163%)
16	36	N/A	N/A	N/A	N/A	N/A	N/A	N/A
29	20	N/A	N/A	N/A	N/A	N/A	N/A	N/A
44	13	N/A	N/A	N/A	N/A	N/A	N/A	N/A
72	8	N/A	N/A	N/A	N/A	N/A	N/A	N/A
143	4	N/A	N/A	N/A	N/A	N/A	N/A	N/A
286	2	N/A	N/A	N/A	N/A	N/A	N/A	N/A
571**(P2N)**	1	5.47	18289	(72%)	1	(0%)	31857	(63%)
571**(N2P)**	1	5.47	18285	(72%)	1	(0%)	31844	(62%)

Tables 4.16, 4.17, 4.18 and 4.19 show the resource utilization and required runtimes for the conversion units over $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{283}}$, $\mathbb{F}_{2^{409}}$ and $\mathbb{F}_{2^{571}}$ respectively. The column values that have been filled with "N/A" designate systems that could not be synthesized due to memory constraints. In each of these cases is is obvious that even if the system could have been synthesized it would have been larger than the full digit width implementation and not a valid option.

For the larger field sizes ($\mathbb{F}_{2^{409}}$ and $\mathbb{F}_{2^{571}}$) the number of valid digit widths was significantly smaller than for the smaller fields. Also in these cases the full digit width implementations required a very significant amount of resources, making them invalid options. This will almost certainly assure that the mixed basis approach is not a valid option for these larger bases as the added runtime for conversion will outweigh the benefit gained by using the normal basis squaring and polynomial basis multiplier. In the case of the normal basis systems this will not have as significant of an impact as basis conversion only needs to be performed once on the final result.

4.2 System Results

The process of comparing the three system approaches is a slightly more complex task than evaluating the efficiencies of the individual hardware units. In the case of a scalar multiplication system various factors come into play, such as parameter settings for each system (i.e. multiplier digit widths and basis conversion digit widths), which affect average runtime. Also, the runtimes of these systems are not constant as computing kP will require a different number of clock cycles depending on the value of k. The reason for this is that factors such as the Hamming weight of RTNAF$_w(k)$ and the placement of nonzero values in RTNAF$_w(k)$ will affect aspects of the scalar multiplication operation such as the number of point additions and $\mathbb{F}_{2^m}$ element squarings required. Therefore, unlike the cases of the individual hardware units, the execution times cannot simply be stated in terms of clock cycles, instead average runtimes must be measured. In order to mitigate the effects

of the Hamming weight and nonzero element locations in RTNAF$_w(k)$, 10,000 element random test vectors were used to calculate the average runtime of a system for a given set of parameters. In order for the comparison between the different approaches to be as accurate as possible one set of test vectors was generated for each field and kept constant for each of the different approaches.

Although the evaluation of the scalar multiplications units is a more complex task, the same metric of efficiency can be used to accurately compare the different system approaches. Overall, the goal of this section, and one of the main goals of this thesis, is to show which of the three approaches is the most efficient in the cases of the five NIST recommended fields, with the limiting factor being the resources available on the FX60. The units used for relative efficiency in this section are (microseconds $\times$ *total number of slices*)$^{-1}$ and, as was the case in the previous section, the "% Slices," "% FFs," and "% LUTs" columns refer to the percentage of the FX60's resources that have been used to implement the system.

4.2.1 $\mathbb{F}_{2^{163}}$ Scalar Multiplication Unit Results

The systems defined over $\mathbb{F}_{2^{163}}$ will be used as a case study to examine the relationship between the different system parameters and the runtime, resource utilization and overall efficiency of the different approaches. $\mathbb{F}_{2_{163}}$ was chosen as the case study basis as it is the smallest of the field recommended by NIST for Koblitz curves and will thus require the least synthesis times for parameter variations of the different systems and will allow the largest range of parameters to be examined on the limited resources of the FX60. The two parameters of interest are the multiplier digit width, designated g, and the conversion unit digit width, designated d.

Polynomial Basis Approach

The following tables detail the resource utilization, execution times and resource efficiency for both the blocking and non-blocking variants of the polynomial basis scalar multiplication unit. In the case of the polynomial basis approach the only relevant system parameter is the polynomial basis multiplier digit width.

As Table 4.20 and Table 4.21 show, for most of the digit widths the difference in slice and LUT utilization between the blocking and non-blocking approach is negligible. The reason for this is that the additional logic required for the non-blocking variant is mainly comprised of additional registers, which are abundant given that the polynomial basis approach does not require a significant amount in the first place. However, for the case of $g = 163$ the resource variance between the blocking and non-blocking systems is significant. The reason for this increased variance is most likely related to the fact that at full digit width the multiplier is implemented as a completely combinational component. The change in the multiplier's implementation did affect the resource utilization in a significant way, however, it is unclear as to exactly why this was the case. What is clear is that the non-blocking approach may not be an attractive option for the full digit width case. In the full digit width case of the non-blocking polynomial basis approach multiplication is not any faster than in the blocking approach and the only significant benefit of the non-blocking approach is the technique used to mitigate the effect of sequential squarings as discussed in Section 4.5.3.

Table 4.20: $\mathbb{F}_{2^{163}}$ Polynomial Basis Blocking Approach Resource Utilization

Param	Slices	% Slices	Regs	% Regs	LUTs	% LUTs
$g = 1$	7144	(28%)	5542	(10%)	10802	(21%)
$g = 2$	7110	(28%)	5502	(10%)	10932	(21%)
$g = 4$	7226	(28%)	5530	(10%)	11242	(22%)
$g = 8$	7820	(30%)	5519	(10%)	12189	(24%)
$g = 17$	8551	(33%)	5516	(10%)	13588	(26%)
$g = 21$	8839	(34%)	5501	(10%)	14165	(28%)
$g = 41$	10396	(41%)	5502	(10%)	17307	(34%)
$g = 82$	12992	(51%)	5747	(11%)	22242	(43%)
$g = 163$	20675	(81%)	6148	(12%)	33414	(66%)

Table 4.21: $\mathbb{F}_{2^{163}}$ Polynomial Basis Non-Blocking Approach Resource Utilization

Param	Slices	% Slices	Regs	% Regs	LUTs	% LUTs
$g = 1$	7340	(29%)	6163	(12%)	10465	(20%)
$g = 2$	7361	(29%)	6157	(12%)	10691	(21%)
$g = 4$	7346	(29%)	6155	(12%)	10734	(21%)
$g = 8$	7844	(31%)	6153	(12%)	11471	(22%)
$g = 17$	8527	(33%)	6209	(12%)	13093	(25%)
$g = 21$	9014	(35%)	6196	(12%)	14101	(27%)
$g = 41$	10613	(41%)	6208	(12%)	17031	(33%)
$g = 82$	13058	(51%)	6297	(12%)	21981	(43%)
$g = 163$	22441	(88%)	6227	(12%)	32745	(64%)

Tables 4.22 and 4.23 show the runtimes for the blocking and non-blocking variants of the polynomial basis approach for a variety of multiplier digit widths. As was expected the non-blocking approach was consistently faster than blocking approach, even in the case of the full digit width multiplier. This shows that there was something to be gained from adding the additional logic required for the non-blocking approach, but does not confirm whether or not using that logic for such a purpose was more efficient.

The runtimes shown correspond to either the double-and-add method for computing scalar multiples on general elliptic curves or the RTNAF$_w$ recoding method for computing scalar multiples on Koblitz curves. The trend that can be seen when examining the RTNAF$_w$ recoding for various widths w is that RTNAF$_5$ is always the fastest. This falls in line with the average Hamming weights of these recoding styles as specified in Table

3.1. The interesting thing about the runtimes is that for the larger digit widths the double-and-add approach is actually faster than the RTNAF$_w$ approach. This is a bit troubling as RTNAF$_w$ should be inherently faster since the Hamming weight is lower and point doublings are replaced with the faster Frobenius map operation. However, the reason behind this situation is the significant time required to perform RTNAF$_w$ recoding. As will be shown later, the cost of performing RTNAF$_w$ recoding is significant, however, this is a constant runtime cost that does not vary with digit width. This is precisely why for the lower digit widths the RTNAF$_w$ approach is faster than the double-and-add method. In the cases of the lower digit widths the time required to compute kP given RTNAF$_w(k)$ has increased enough that the time required to compute the recoding is less significant. In these situations the benefit of the RTNAF$_w$ technique is apparent.

Table 4.22: $\mathbb{F}_{2^{163}}$ Polynomial Basis Blocking Approach Execution Time (msec)

Param	**Dbl-n-Add**	**RTNAF$_2$**	**RTNAF$_3$**	**RTNAF$_4$**	**RTNAF$_5$**	**RTNAF$_6$**
$g = 1$	2.3636	1.5491	1.4659	1.3866	1.3636	1.5744
$g = 2$	1.3832	1.1510	1.0725	1.0124	0.9901	1.1109
$g = 4$	0.8869	0.9495	0.8734	0.8230	0.8011	0.8763
$g = 8$	0.6449	0.8512	0.7762	0.7307	0.7089	0.7619
$g = 17$	0.5117	0.7971	0.7228	0.6798	0.6582	0.6990
$g = 21$	0.4875	0.7873	0.7131	0.6706	0.6490	0.6875
$g = 41$	0.4391	0.7676	0.6937	0.6521	0.6306	0.6647
$g = 82$	0.4149	0.7578	0.6840	0.6429	0.6213	0.6532
$g = 163$	0.3907	0.7480	0.6743	0.6337	0.6121	0.6418

Table 4.23: $\mathbb{F}_{2^{163}}$ Polynomial Basis Non-Blocking Approach Execution Time (msec)

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1$	2.2556	1.4919	1.4122	1.3337	1.3087	1.5066
$g = 2$	1.2752	1.0938	1.0188	0.9595	0.9353	1.0432
$g = 4$	0.7789	0.8923	0.8196	0.7702	0.7463	0.8086
$g = 8$	0.5368	0.7940	0.7244	0.6803	0.6569	0.6978
$g = 17$	0.4133	0.7463	0.6782	0.6367	0.6136	0.6444
$g = 21$	0.3987	0.7418	0.6729	0.6313	0.6081	0.6376
$g = 41$	0.3823	0.7393	0.6666	0.6241	0.6006	0.6282
$g = 82$	0.3741	0.7380	0.6635	0.6205	0.5968	0.6235
$g = 163$	0.3739	0.7378	0.6633	0.6204	0.5966	0.6233

Table 4.24 and 4.25 show the efficiencies for the blocking and non-blocking polynomial basis approaches. As was to be expected the non-blocking approach is almost always more efficient than the blocking approach. The caveat to this statement is that in the case of the full digit width, even though the non-blocking system is faster, the additional resources required were not used efficiently enough.

Table 4.24: $\mathbb{F}_{2^{163}}$ Polynomial Basis Blocking Approach Efficiency

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1$	59.22	90.36	95.49	100.95	102.65	88.91
$g = 2$	101.68	122.20	131.14	138.92	142.05	126.61
$g = 4$	156.04	145.75	158.45	168.15	172.75	157.92
$g = 8$	198.29	150.23	164.75	175.01	180.39	167.84
$g = 17$	228.54	146.71	161.79	172.03	177.67	167.30
$g = 21$	232.07	143.70	158.65	168.71	174.32	164.56
$g = 41$	219.06	125.31	138.66	147.51	152.54	144.71
$g = 82$	185.52	101.57	112.53	119.72	123.89	117.84
$g = 163$	123.80	64.66	71.73	76.33	79.02	75.36

Table 4.25: $\mathbb{F}_{2^{163}}$ Polynomial Basis Non-Blocking Approach Efficiency

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1$	60.40	91.32	96.47	102.15	104.10	90.42
$g = 2$	106.53	124.2	133.34	141.59	145.25	130.23
$g = 4$	174.77	152.56	166.09	176.74	182.40	168.35
$g = 8$	237.49	160.56	175.99	187.40	194.07	182.70
$g = 17$	283.75	157.14	172.92	184.19	191.13	181.99
$g = 21$	278.25	149.55	164.87	175.73	182.43	173.99
$g = 41$	246.47	127.45	141.35	150.98	156.88	149.99
$g = 82$	204.71	103.77	115.42	123.42	128.32	122.83
$g = 163$	119.18	60.40	67.18	71.83	74.69	71.49

Normal Basis Approach

The next system to be examined is the normal basis approach. In the case of this system in addition to varying the multiplier digit width, the basis conversion digit width could also have been varied. However, as the resources allowed for it, the conversion unit digit width was kept at a maximum. Tables 4.26 and 4.27 show the resource utilization for both the blocking and non-blocking variants of the normal basis approach. It is important to note that resource utilization results are shown for the full digit width case, $g = 163$, even though the system did not meet timing requirements. The resource utilization is shown to give an idea of what would be required for the full digit width case. It is also important to note that even decreasing the conversion digit width d to 1 did not help alleviate these timing problems.

In comparing the resource requirements of the blocking and non-blocking approaches we can observe that, as was the case with the polynomial basis approach, the resource utilization for these two approaches is almost identical. The non-blocking approach does require more registers than the blocking, but this is to be expected as the non-blocking approach requires these resources to store intermediate pipelined values. One interesting observation in comparing the resource utilization of these two approaches is that although the resource utilization is close, unlike in the case of the polynomial basis approach, the non-blocking approach actually requires slightly fewer slices and LUTs for the majority

of the scenarios. It is hard to say exactly why this is the case, but it may have been that introducing the additional logic needed for the non-blocking approach actually aided in resource consolidation. One possible reason we see such an effect in the normal basis approach and not in the polynomial basis approach is that in the case of the normal basis approach the non-blocking variant only affects multiplication. The normal basis squaring unit can complete up to m consecutive squarings in a single clock cycle and unlike the polynomial basis squaring unit, no additional steps could have be taken in the non-blocking variant to make this any faster.

Table 4.26: $\mathbb{F}_{2^{163}}$ Normal Basis Blocking Approach Resource Utilization

Param	Slices	% Slices	Regs	% Regs	LUTs	% LUTs
$g = 1, d = 163$	9680	(38%)	5093	(10%)	15576	(30%)
$g = 2, d = 163$	9805	(38%)	5098	(10%)	15815	(31%)
$g = 4, d = 163$	10024	(39%)	5097	(10%)	16296	(32%)
$g = 8, d = 163$	10339	(40%)	5092	(10%)	17021	(33%)
$g = 17, d = 163$	10877	(43%)	5082	(10%)	18587	(36%)
$g = 21, d = 163$	11117	(43%)	5078	(10%)	19134	(37%)
$g = 41, d = 163$	13261	(52%)	5078	(10%)	23398	(46%)
$g = 82, d = 163$	19014	(75%)	5224	(10%)	32201	(63%)
$g = 163, d = 163$	25278	(99%)	5408	(10%)	48051	(95%)

Table 4.27: $\mathbb{F}_{2^{163}}$ Normal Basis Non-Blocking Approach Resource Utilization

Param	Slices	% Slices	Regs	% Regs	LUTs	% LUTs
$g = 1, d = 163$	9595	(37%)	5261	(10%)	15564	(30%)
$g = 2, d = 163$	9707	(38%)	5260	(10%)	15805	(31%)
$g = 4, d = 163$	9923	(39%)	5259	(10%)	16286	(32%)
$g = 8, d = 163$	10272	(40%)	5285	(10%)	17011	(33%)
$g = 17, d = 163$	10904	(43%)	5244	(10%)	18470	(36%)
$g = 21, d = 163$	11220	(44%)	5245	(10%)	19033	(37%)
$g = 41, d = 163$	13574	(53%)	5262	(10%)	23537	(46%)
$g = 82, d = 163$	18954	(74%)	5243	(10%)	31927	(63%)
$g = 163, d = 163$	25278	(99%)	5497	(10%)	48225	(95%)

Tables 4.28 and 4.29 show the execution times for the blocking and non-blocking variants of the normal basis approach. As was mentioned earlier, for the full digit width case, $g = 163$, the design did not meet the necessary timing constraints and, therefore, no valid timing data could be captured. The inability to capture the data for the full digit width case is definitely a loss as it most likely would have shown that the blocking and non-blocking approaches had the same, or at least nearly the same, execution time for these system parameters. The reason that this is believed to be true is that in the normal basis approach the non-blocking variant only affects multiplication, and if the multiplier can produce a product in a single clock cycle, then there is little to be gained from the simple pipelining implemented in this technique.

Overall the results shown in these tables are what was to be expected. The non-blocking approach provided an advantage over the blocking approach as multi-cycle multiplications are pipelined, which resulted in an overall faster execution time. The normal basis approach offers no additional insight into the comparison of the various RTNAF$_w$ conversions and the double-and-add approach, as these results mirror those captured for the polynomial basis approach. Again RTNAF$_5$ proves to be the fastest and we see the same shift as the double-and-add method proves to be faster than RTNAF$_w$ for the larger digit width systems.

Table 4.28: $\mathbb{F}_{2^{163}}$ Normal Basis Blocking Approach Execution Time (msec)

Param	**Dbl-n-Add**	**RTNAF$_2$**	**RTNAF$_3$**	**RTNAF$_4$**	**RTNAF$_5$**	**RTNAF$_6$**
$g = 1, d = 163$	2.3623	1.5477	1.4638	1.3819	1.3552	1.5618
$g = 2, d = 163$	1.3697	1.1447	1.0656	1.0031	0.9771	1.0927
$g = 4, d = 163$	0.8735	0.9432	0.8664	0.8138	0.7881	0.8581
$g = 8, d = 163$	0.6314	0.8449	0.7693	0.7214	0.6959	0.7437
$g = 17, d = 163$	0.4983	0.7908	0.7159	0.6706	0.6452	0.6807
$g = 21, d = 163$	0.4741	0.7810	0.7062	0.6613	0.6360	0.6693
$g = 41, d = 163$	0.4257	0.7614	0.6867	0.6429	0.6176	0.6464
$g = 82, d = 163$	0.4014	0.7515	0.6770	0.6336	0.6083	0.6350
$g = 163, d = 163$	N/A	N/A	N/A	N/A	N/A	N/A

Table 4.29: $\mathbb{F}_{2^{163}}$ Normal Basis Non-Blocking Approach Execution Time (msec)

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1, d = 163$	2.2542	1.4938	1.4074	1.3290	1.3022	1.4959
$g = 2, d = 163$	1.2617	1.0908	1.0092	0.9502	0.9242	1.0268
$g = 4, d = 163$	0.7655	0.8893	0.8101	0.7608	0.7352	0.7922
$g = 8, d = 163$	0.5234	0.7910	0.7150	0.6712	0.6460	0.6818
$g = 17, d = 163$	0.4014	0.7444	0.6697	0.6283	0.6035	0.6293
$g = 21, d = 163$	0.3932	0.7431	0.6666	0.6248	0.5997	0.6246
$g = 41, d = 163$	0.3768	0.7405	0.6603	0.6176	0.5921	0.6152
$g = 82, d = 163$	0.3727	0.7398	0.6586	0.6157	0.5902	0.6127
$g = 163, d = 163$	N/A	N/A	N/A	N/A	N/A	N/A

Lastly Tables 4.30 and 4.31 show the efficiencies of the blocking and non-blocking variants of the normal basis approach. In all of the cases shown, given that the results for the $g = 163$ could not be captured due to the inability to meet timing constraints, the non-blocking approach has proven to be more efficient than the blocking approach. This is to be expected as not only did the non-blocking approach consistently have a faster runtime, but, surprisingly, it almost always required fewer resources.

Table 4.30: $\mathbb{F}_{2^{163}}$ Normal Basis Blocking Approach Efficiency

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1, d = 163$	43.73	66.75	70.57	74.76	76.23	66.15
$g = 2, d = 163$	74.46	89.10	95.71	101.67	104.38	93.34
$g = 4, d = 163$	114.21	105.77	115.14	122.59	126.58	116.26
$g = 8, d = 163$	153.19	114.48	125.73	134.07	138.99	130.05
$g = 17, d = 163$	184.50	116.26	128.42	137.10	142.49	135.06
$g = 21, d = 163$	189.73	115.18	127.38	136.02	141.43	134.4
$g = 41, d = 163$	177.14	99.04	109.81	117.30	122.10	116.66
$g = 82, d = 163$	131.02	69.98	77.69	83.01	86.46	82.82
$g = 163, d = 163$	N/A	N/A	N/A	N/A	N/A	N/A

Table 4.31: $\mathbb{F}_{2^{163}}$ Normal Basis Non-Blocking Approach Efficiency

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1, d = 163$	46.23	69.77	74.05	78.42	80.03	69.67
$g = 2, d = 163$	81.65	94.44	102.08	108.42	111.47	100.33
$g = 4, d = 163$	131.65	113.32	124.4	132.46	137.07	127.21
$g = 8, d = 163$	186.00	123.07	136.16	145.04	150.70	142.79
$g = 17, d = 163$	228.47	123.2	136.94	145.96	151.96	145.73
$g = 21, d = 163$	226.67	119.94	133.7	142.65	148.62	142.69
$g = 41, d = 163$	195.52	99.49	111.57	119.28	124.42	119.75
$g = 82, d = 163$	141.56	71.32	80.11	85.69	89.39	86.11
$g = 163, d = 163$	N/A	N/A	N/A	N/A	N/A	N/A

Mixed Basis Approach

Lastly the performance of the mixed basis approach will be evaluated. The variable parameters for the mixed basis systems are multiplier digit width and conversion unit digit width. It should be noted that the single conversion unit digit width parameter applies to both the polynomial to normal basis conversion unit as well as the normal basis to polynomial basis conversion unit. The reason for this is that both conversion units are executed nearly the same amount when computing a scalar multiple and, therefore, there was no reason to set the two conversion units with different digit widths. As was the case with the normal basis approach the conversion unit digit width was kept at a maximum as the resources allowed for it. This is a bit more significant in the case of the mixed basis approach as basis conversion occurs rather frequently and the cost of lowering the conversion digit width would have been rather significant as the next reasonably sized conversion unit takes 10 times longer to complete, as shown in Table 4.15.

Tables 4.32 and 4.33 show the resource utilization required for the blocking and non-blocking variants of the mixed basis approach. As was the case with the other two approaches the resource utilization for these two techniques was almost identical. In these cases the overall slice and LUT utilization were sometimes greater for the blocking approach and sometimes greater for the non-blocking approach, however, the non-blocking approach consistently required additional registers, as was to be expected.

Table 4.32: $\mathbb{F}_{2^{163}}$ Mixed Basis Blocking Approach Resource Utilization

Param	Slices	% Slices	Regs	% Regs	LUTs	% LUTs
$g = 1, d = 163$	9999	(39%)	4985	(9%)	16540	(32%)
$g = 2, d = 163$	9968	(39%)	4979	(9%)	16662	(32%)
$g = 4, d = 163$	10139	(40%)	4981	(9%)	17009	(33%)
$g = 8, d = 163$	10480	(41%)	4995	(9%)	17438	(34%)
$g = 17, d = 163$	11549	(45%)	4998	(9%)	19439	(38%)
$g = 21, d = 163$	11812	(46%)	4993	(9%)	19997	(39%)
$g = 41, d = 163$	13288	(52%)	4988	(9%)	23164	(45%)
$g = 82, d = 163$	15527	(61%)	4987	(9%)	27817	(55%)
$g = 163, d = 163$	24946	(98%)	4987	(9%)	38500	(76%)

Table 4.33: $\mathbb{F}_{2^{163}}$ Mixed Basis Non-Blocking Approach Resource Utilization

Param	Slices	% Slices	Regs	% Regs	LUTs	% LUTs
$g = 1, d = 163$	9910	(39%)	5147	(10%)	16449	(32%)
$g = 2, d = 163$	10078	(39%)	5149	(10%)	16719	(33%)
$g = 4, d = 163$	10214	(40%)	5149	(10%)	17013	(33%)
$g = 8, d = 163$	10638	(42%)	5158	(10%)	17598	(34%)
$g = 17, d = 163$	11520	(45%)	5161	(10%)	19454	(38%)
$g = 21, d = 163$	11812	(46%)	5154	(10%)	19988	(39%)
$g = 41, d = 163$	13372	(52%)	5149	(10%)	23154	(45%)
$g = 82, d = 163$	15682	(62%)	5149	(10%)	27828	(55%)
$g = 163, d = 163$	24882	(98%)	5141	(10%)	38262	(75%)

The execution times for the blocking and non-blocking mixed basis systems are shown in Tables 4.34 and 4.35. The runtimes for the non-blocking approach were consistently faster than those of the blocking, even in the case of $g = 163$. This is surprising as when $g = 163$ the multiplication unit only required a single clock cycle to complete, and the capability to pipeline multiplication is the primary benefit of the non-blocking system. There were, however, minor adjustments made in the point addition functions for the non-blocking approach and it is most likely that these adjustments were significant enough to result in the non-blocking approach being faster even in the $g = 163$ case.

Table 4.34: $\mathbb{F}_{2^{163}}$ Mixed Basis Blocking Approach Execution Time (msec)

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1, d = 163$	2.3636	1.5798	1.4860	1.4002	1.3707	1.5774
$g = 2, d = 163$	1.3832	1.1817	1.0926	1.0260	0.9973	1.1140
$g = 4, d = 163$	0.8869	0.9802	0.8935	0.8366	0.8083	0.8794
$g = 8, d = 163$	0.6449	0.8819	0.7963	0.7443	0.7161	0.7650
$g = 17, d = 163$	0.5117	0.8278	0.7429	0.6934	0.6654	0.7021
$g = 21, d = 163$	0.4875	0.81805	0.7332	0.6842	0.6562	0.6906
$g = 41, d = 163$	0.4391	0.7983	0.7138	0.6657	0.6377	0.6677
$g = 82, d = 163$	0.4149	0.7885	0.7041	0.6565	0.6285	0.6563
$g = 163, d = 163$	0.3907	0.7787	0.6943	0.6473	0.6193	0.6449

Table 4.35: $\mathbb{F}_{2^{163}}$ Mixed Basis Non-Blocking Approach Execution Time (msec)

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1, d = 163$	2.2556	1.5195	1.4296	1.3472	1.3177	1.5115
$g = 2, d = 163$	1.2752	1.1213	1.0362	0.9731	0.9443	1.0481
$g = 4, d = 163$	0.7789	0.9198	0.8371	0.7837	0.7553	0.8135
$g = 8, d = 163$	0.5368	0.8215	0.7419	0.6938	0.6659	0.7028
$g = 17, d = 163$	0.4133	0.7739	0.6957	0.6502	0.6226	0.6493
$g = 21, d = 163$	0.3987	0.7694	0.6903	0.6448	0.6171	0.6425
$g = 41, d = 163$	0.3823	0.7668	0.6840	0.6376	0.6096	0.6331
$g = 82, d = 163$	0.3741	0.7655	0.6809	0.6341	0.6058	0.6284
$g = 163, d = 163$	0.3739	0.7653	0.6807	0.6339	0.6056	0.6282

Finally Tables 4.36 and 4.37 show the efficiencies for the blocking and non-blocking mixed basis systems. Based on the prior data it should be of little surprise that the non-blocking approach was more efficient than the blocking approach for the cases of $g \leq 82$. However, it is a surprise to see that even in the case of $g = 163$, the non-blocking approach is still more efficient. This does make sense when looking at the resource utilization and runtimes for this scenario as the non-blocking approach actually utilized fewer total slices and still maintained a faster runtime. It can only be assumed that the addition of the non-blocking approach somehow aided in resource consolidation for the mixed basis approach and that is why we see these results.

Table 4.36: $\mathbb{F}_{2^{163}}$ Mixed Basis Blocking Approach Efficiency

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1, d = 163$	42.31	63.31	67.30	71.43	72.96	63.40
$g = 2, d = 163$	72.53	84.90	91.82	97.78	100.59	90.05
$g = 4, d = 163$	111.21	100.62	110.39	117.89	122.02	112.15
$g = 8, d = 163$	147.96	108.20	119.83	128.20	133.25	124.73
$g = 17, d = 163$	186.48	115.27	128.44	137.61	143.40	135.91
$g = 21, d = 163$	173.66	103.49	115.47	123.74	129.02	122.59
$g = 41, d = 163$	171.39	94.27	105.43	113.05	118.01	112.71
$g = 82, d = 163$	155.23	81.68	91.47	98.10	102.47	98.13
$g = 163, d = 163$	102.60	51.48	57.74	61.93	64.73	62.16

Table 4.37: $\mathbb{F}_{2^{163}}$ Mixed Basis Non-Blocking Approach Efficiency

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
$g = 1, d = 163$	44.74	66.41	70.58	74.90	76.58	66.76
$g = 2, d = 163$	77.81	88.49	95.76	101.97	105.08	94.67
$g = 4, d = 163$	125.70	106.44	116.96	124.93	129.62	120.35
$g = 8, d = 163$	175.12	114.43	126.71	135.49	141.17	133.75
$g = 17, d = 163$	210.03	112.17	124.77	133.51	139.42	133.69
$g = 21, d = 163$	212.34	110.03	122.64	131.30	137.19	131.77
$g = 41, d = 163$	195.61	97.53	109.33	117.29	122.68	118.12
$g = 82, d = 163$	170.46	83.30	93.65	100.56	105.26	101.48
$g = 163, d = 163$	107.49	52.51	59.04	63.40	66.36	63.98

$\mathbb{F}_{2^{163}}$ System Approach Comparisons

The most effective way to compare the three system approaches is to analyze the efficiencies of the RTNAF$_w$ encoding methods and double-and-add method for scalar point multiplication on Koblitz curves. Figure 4.11 shows the efficiencies for RTNAF$_2$ encoding for both the blocking and non-blocking variants of the three system approaches over a variety of digit widths. One of the first things to be noticed is that digit widths $g = 8$ and $g = 17$ appear to produce the most efficient systems. This appears to be the turning point at which the additional resource costs for a faster multiplier outweigh the increase in speed. If these systems were being parameterized to share resources with other units on the same chip, these digit widths would be ideal.

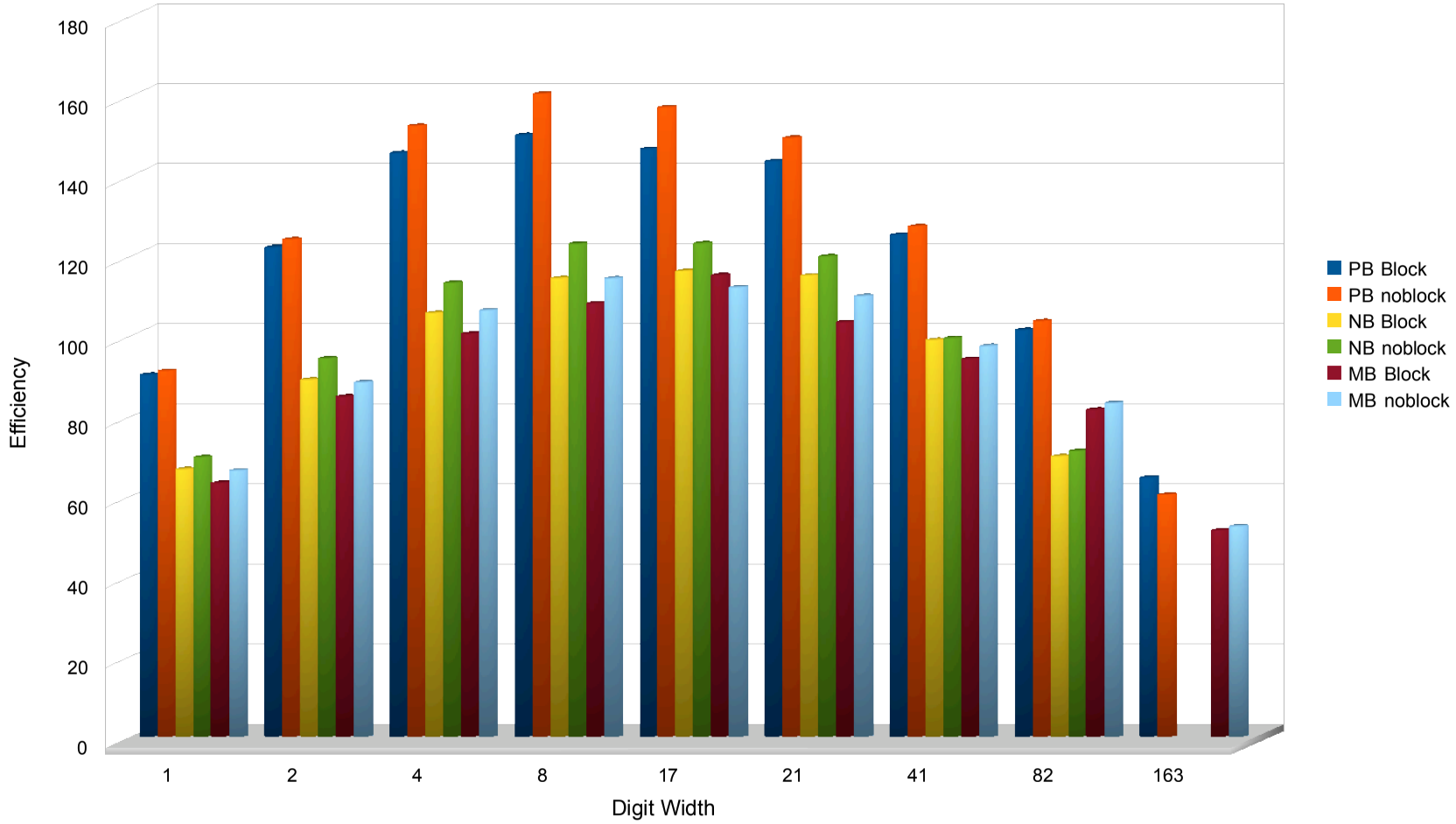


Figure 4.11: RTNAF$_2$ Efficiency Comparisons for $\mathbb{F}_{2^{163}}$

In comparing the different system approaches we can see that in each of the cases the polynomial basis approach is the most efficient by a significant amount. It is surprising that even the blocking polynomial basis system is more efficient than the non-blocking normal basis approach. Apparently the steps taken to minimize the impact of consecutive squarings in the polynomial basis system did help increase efficiency, but they were not entirely necessary in order to be more efficient than the normal basis approach. The normal basis approach ranks second overall in terms of efficiency, however, in the case of $g = 82$ the mixed basis approach does overtake the normal basis approach. This can be attributed to the growth in resource utilization for the normal basis multiplier shown in Table 4.2. It would seem that there is a point at which the efficiency of the normal basis multiplier does not increase significantly enough and this allows the mixed basis approach to be more efficient, despite the requirement for frequent conversion between the two bases in the mixed basis approach.

While it is not too surprising that the mixed basis approach almost always ranked in last, it is surprising that normal basis approach did not contend with the blocking polynomial basis approach. One reason for this may be the normal basis squaring unit. Creating a unit that can cyclically shift a 163-bit string anywhere from 1 to 163 times in a single clock cycle does not require a very significant amount of resources, but when it has been replicated numerous times the resource utilization and routing adds up. More time and research could have been invested in either reusing squaring units for the normal basis approach or examining the tradeoff in other approaches, such as a unit that can only perform up to 32 squarings in a single clock cycle.

Figures 4.12, 4.13. 4.14 and 4.15 mirror the results shown in Figure 4.11. The only difference between these graphs are the levels off efficiency for each digit width for each graph. This variance in efficiency, with RTNAF$_5$ being the most efficient of the group, is in line with the data on average Hamming weights provided in Table 3.1.

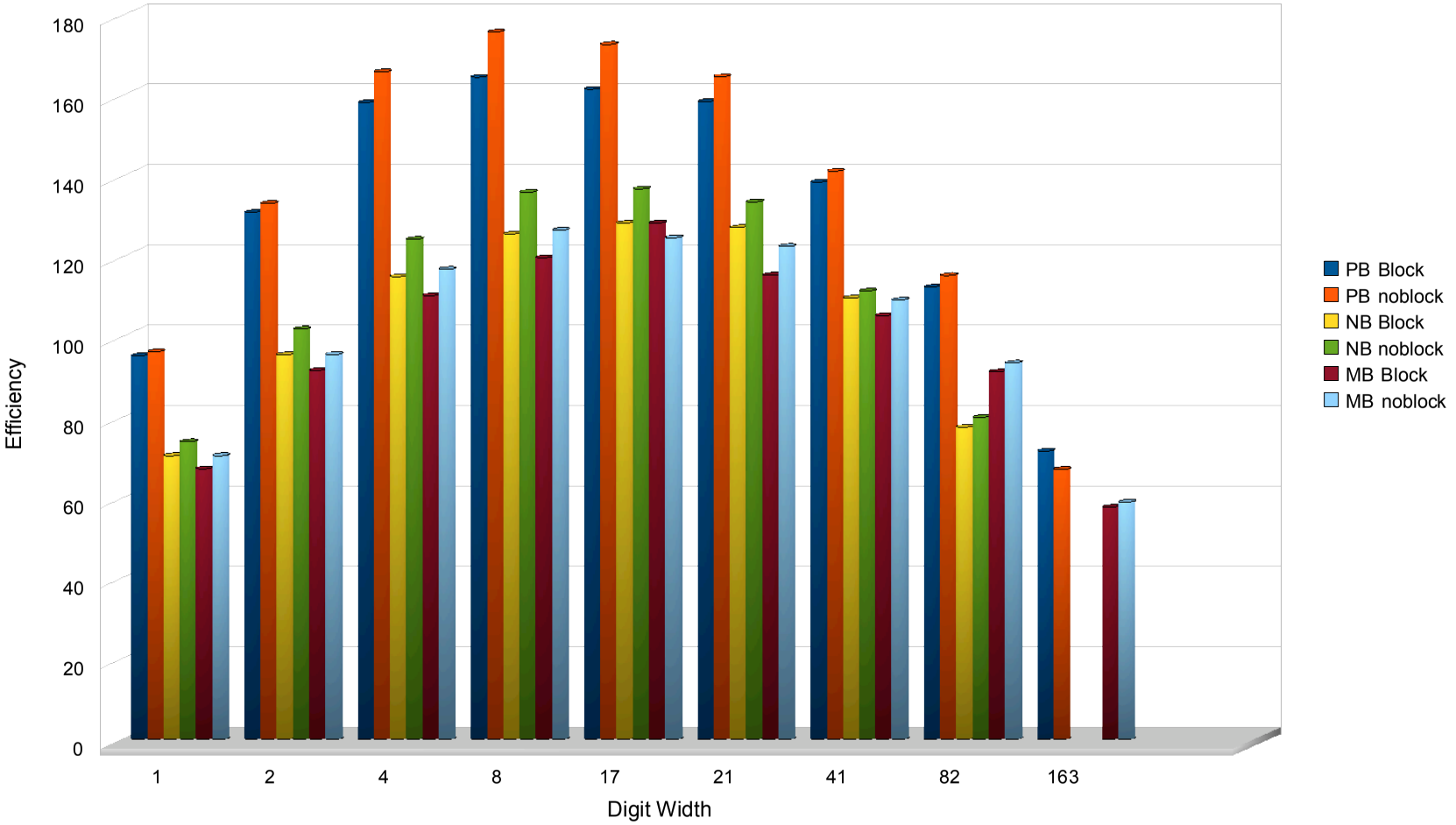


Figure 4.12: RTNAF$_3$ Efficiency Comparisons for $\mathbb{F}_{2^{163}}$

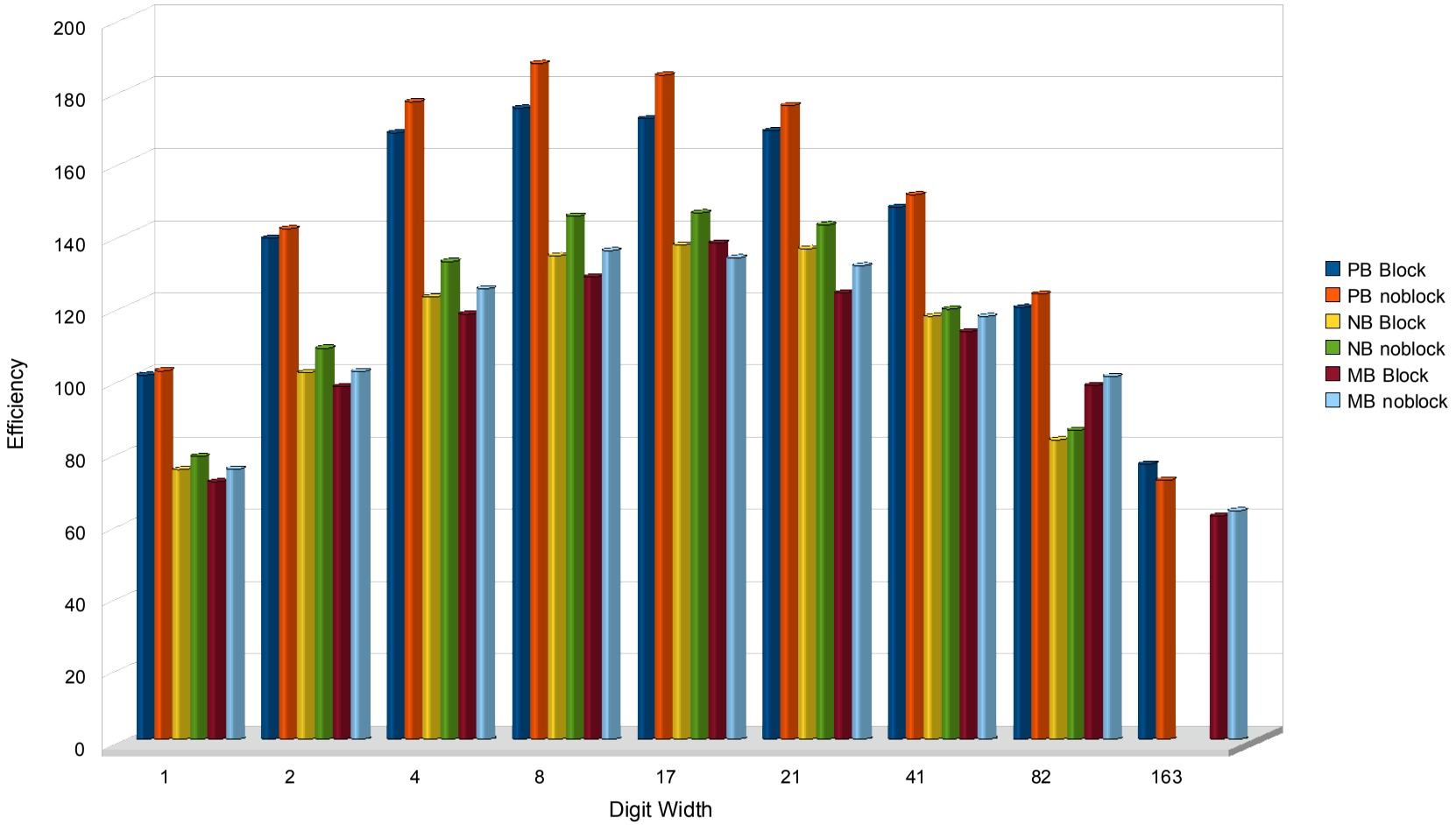


Figure 4.13: RTNAF$_4$ Efficiency Comparisons for $\mathbb{F}_{2^{163}}$

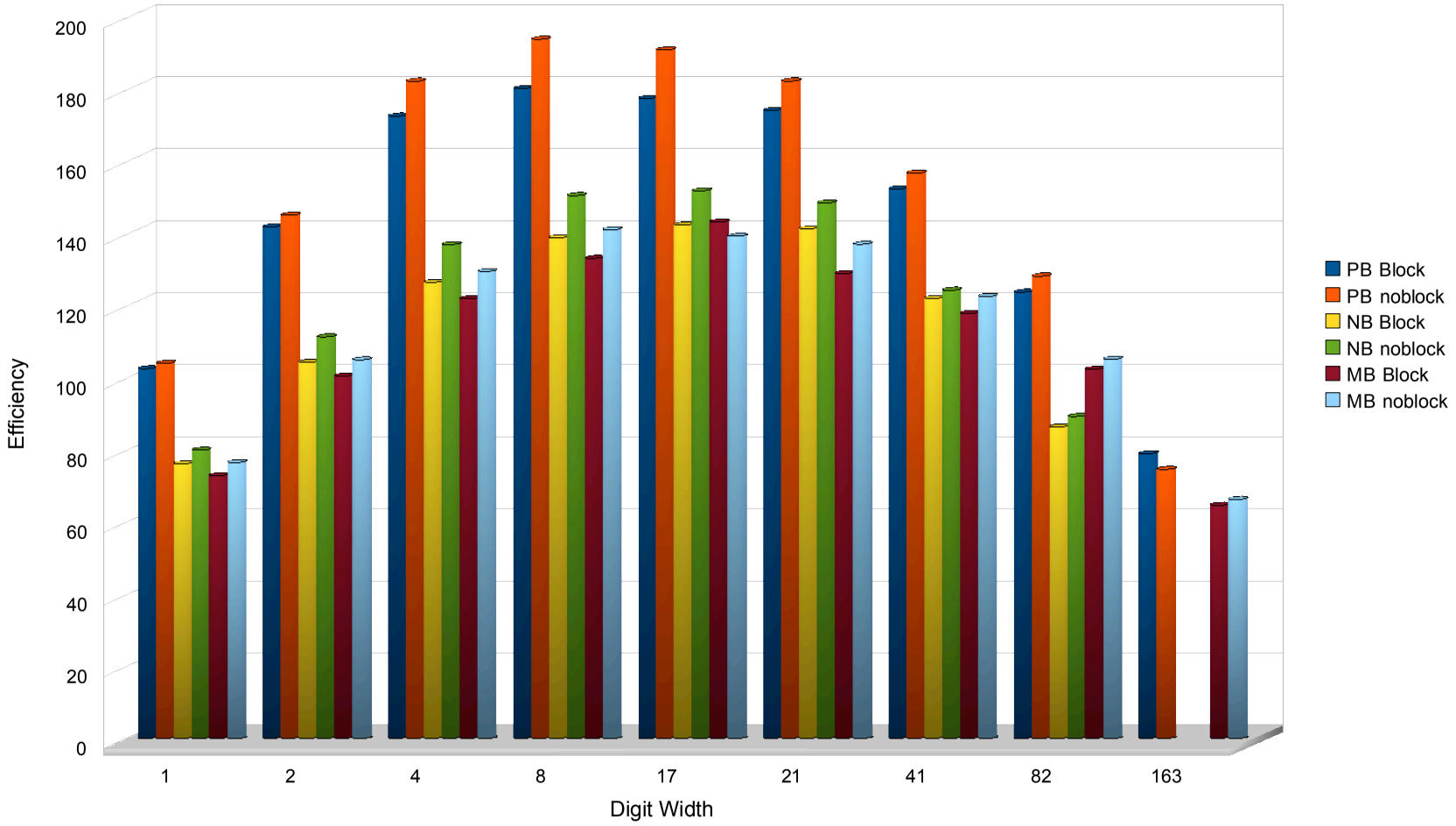


Figure 4.14: RTNAF$_5$ Efficiency Comparisons for $\mathbb{F}_{2^{163}}$

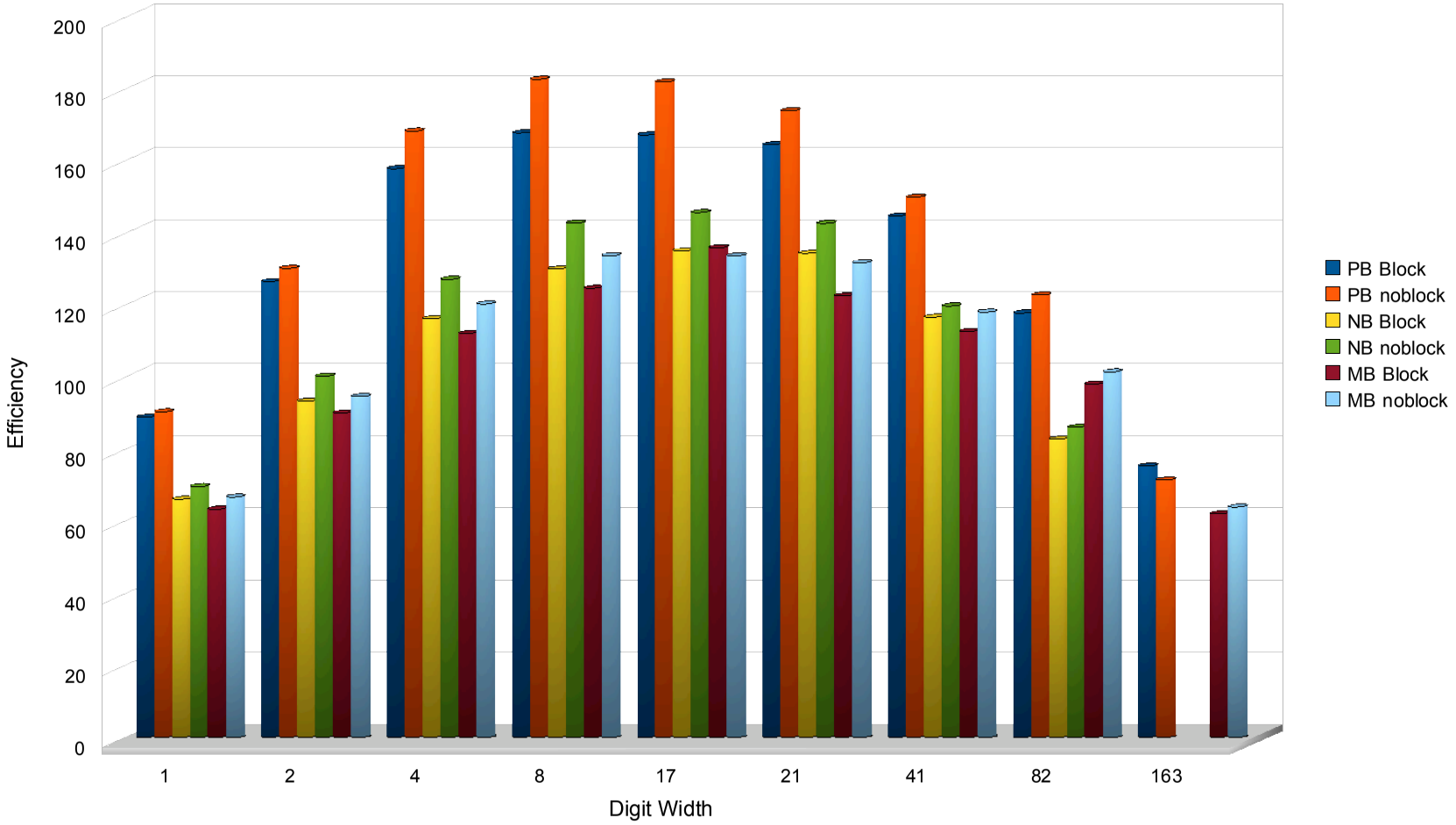


Figure 4.15: RTNAF$_6$ Efficiency Comparisons for $\mathbb{F}_{2^{163}}$

Finally, Figure 4.16 shows the efficiency data corresponding to the double-and-add approach for the three system approaches over a variety of digit widths. The data presented here is almost exactly the same as that presented for RTNAF$_w$ except that there is a greater disparity between the blocking and non-blocking approach. The reason for this is most likely that more point doublings are required in the double-and-add method and this emphasized the benefit of the non-blocking approach.

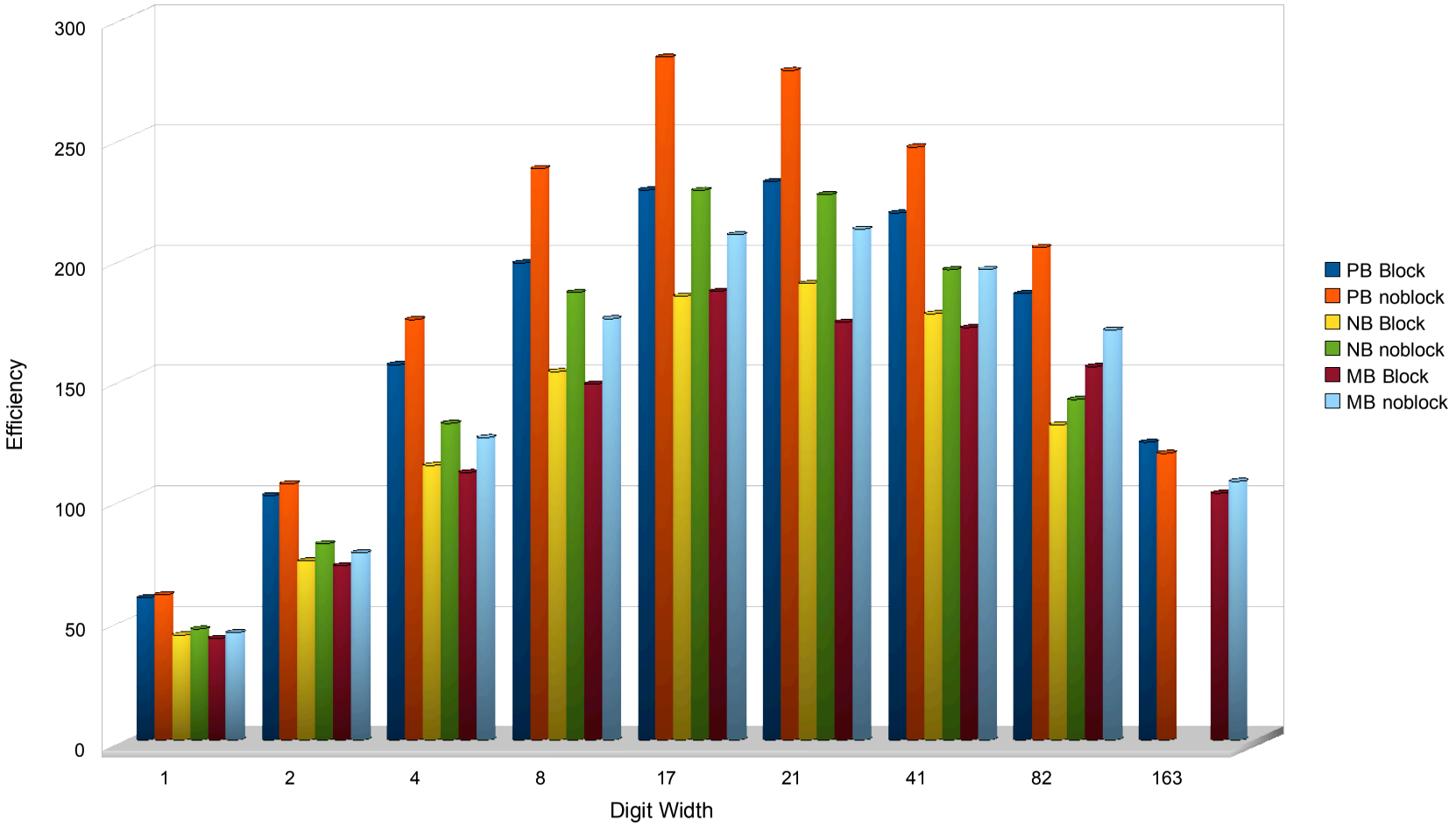


Figure 4.16: Double-and-Add Efficiency Comparisons for $\mathbb{F}_{2^{163}}$

4.2.2 Overall System Results

Lastly the fastest achievable scalar multiplication unit for each basis and system approach will be examined. It should be noted that the while the Virtex-4 is a high end FPGA, the FX60 is the medium sized model and will not allow for the larger systems to be implemented with ideal parameter settings.

Table 4.38: $\mathbb{F}_{2^{163}}$ Resource Utilization

System Settings	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
Block PB: $g = 163$	20675	(81%)	6148	(12%)	33414	(66%)
NonBlock PB: $g = 163$	22441	(88%)	6227	(12%)	32745	(64%)
Block NB: $g = 82, d = 163$	19014	(75%)	5224	(10%)	32201	(63%)
NonBlock NB: $g = 82, d = 163$	18954	(74%)	5243	(10%)	31927	(63%)
Block MB: $g = 163, d = 163$	24946	(98%)	4987	(9%)	38500	(76%)
NonBlock MB: $g = 163, d = 163$	24882	(98%)	5141	(10%)	38262	(75%)

Table 4.39: $\mathbb{F}_{2^{163}}$ Execution Time (msec)

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
Block PB: $g = 163$	0.3907	0.7480	0.6743	0.6337	0.6121	0.6418
NonBlock PB: $g = 163$	0.3739	0.7378	0.6633	0.6204	0.5966	0.6233
Block NB: $g = 82, d = 163$	0.4014	0.7515	0.6770	0.6336	0.6083	0.6350
NonBlock NB: $g = 82, d = 163$	0.3727	0.7398	0.6586	0.6157	0.5902	0.6127
Block MB: $g = 163, d = 163$	0.3907	0.7787	0.6943	0.6473	0.6193	0.6449
NonBlock MB: $g = 163, d = 163$	0.3739	0.7653	0.6807	0.6339	0.6056	0.6282

Tables 4.38 and 4.39 show the resource utilization and execution times for the fastest $\mathbb{F}_{2^{163}}$ scalar multiplication systems that could fit on the FX60. The $\mathbb{F}_{2^{163}}$ basis is the smallest of all the NIST bases and, therefore, the most likely to support full digit width systems for each approach. The FX60 should easily have been able to support each of these systems at full digit width, however, the normal basis approach needed to be scaled back in terms of multiplier digit width to $g = 82$ in order to meet timing constraints. Overall, even though the system needed to be scaled back, the non-blocking normal basis approach proved to be the fastest system overall. In the end, given that resources are available for ideal parameter settings this data shows that the normal basis approach is faster than the other two approaches. The steps taken to improve the polynomial basis approach did help the system achieve a faster runtime, as comparing the blocking and non-blocking approach show, but these were not enough to run faster than the normal basis system. Of course, the resource requirements for a normal basis system were significantly larger than for a polynomial basis system, and as the data in the previous section showed, the polynomial basis approach was the more efficient approach.

Table 4.40: $\mathbb{F}_{2^{233}}$ Resource Utilization

System Settings	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
Block PB: $g = 117$	22100	(87%)	6751	(13%)	37075	(73%)
NonBlock PB: $g = 117$	24373	(96%)	7309	(14%)	36935	(73%)
Block NB: $g = 34, d = 8$	21801	(86%)	5664	(11%)	38353	(75%)
NonBlock NB: $g = 34, d = 8$	21339	(84%)	6179	(12%)	37585	(74%)
Block MB: $g = 87, d = 233$	21818	(86%)	5064	(10%)	40084	(79%)
NonBlock MB: $g = 87, d = 233$	22316	(88%)	5299	(10%)	40678	(80%)

Table 4.41: $\mathbb{F}_{2^{233}}$ Execution Time (msec)

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
Block PB: $g = 117$	0.5900	1.3176	1.1868	1.1136	1.0715	1.1573
NonBlock PB: $g = 117$	0.5285	1.2854	1.1570	1.0824	1.0368	1.1097
Block NB: $g = 34, d = 8$	0.6575	1.3441	1.2118	1.1328	1.0833	1.1688
NonBlock NB: $g = 34, d = 8$	0.5594	1.2898	1.1675	1.0945	1.0473	1.1197
Block MB: $g = 87, d = 233$	0.6073	1.3661	1.2223	1.1390	1.0863	1.1692
NonBlock MB: $g = 87, d = 233$	0.5365	1.3256	1.1912	1.1114	1.0601	1.1328

Tables 4.40 and 4.41 show the resource utilization and execution results for the $\mathbb{F}_{2^{233}}$ systems. In this case the polynomial basis approach had the fastest execution times as the normal basis approach has issues meeting timing constraints for faster systems. Overall, the timing constraint issue became more prevalent as the basis size was increased. For the next basis, $\mathbb{F}_{2^{283}}$ (Tables 4.42 and 4.43), a normal basis system with even the most frugal parameter settings could not be synthesized that met timing. Moving onto the next basis of $\mathbb{F}_{2^{409}}$ (Tables 4.44 and 4.45) the mixed basis approach could not meet timing even with the smallest parameter settings. Lastly Tables 4.46 and 4.47 show the resource and execution time results for the polynomial basis approach in the case of the largest NIST basis of $\mathbb{F}_{2^{571}}$. The issue of not meeting timing constraints grew from being a minor annoyance to a crippling error. Since this issue was prevalent primarily in the normal basis and mixed basis approaches, it is believed that it was mostly related to the normal basis squaring unit. The issue of timing constraint errors will be discussed further in the Conclusion.

Table 4.42: $\mathbb{F}_{2^{283}}$ Resource Utilization

System Settings	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
Block PB: $g = 95$	22546	(89%)	6881	(13%)	40808	(80%)
NonBlock PB: $g = 95$	22827	(90%)	8152	(16%)	40432	(79%)
Block NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
Block MB: $g = 71, w = 8$	21193	(83%)	5944	(11%)	38024	(75%)
NonBlock MB: $g = 71, w = 8$	21300	(84%)	6242	(12%)	37717	(74%)

Table 4.43: $\mathbb{F}_{2^{283}}$ Execution Time (msec)

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
Block PB: $g = 95$	0.7447	1.9303	1.7520	1.6499	1.5881	1.6992
NonBlock PB: $g = 95$	0.6585	1.8838	1.7104	1.6079	1.5419	1.6352
Block NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
Block MB: $g = 71, w = 8$	0.7657	2.1779	1.9390	1.7959	1.7006	1.8231
NonBlock MB: $g = 71, w = 8$	0.6684	2.1239	1.8932	1.7557	1.6630	1.7733

Table 4.44: $\mathbb{F}_{2^{409}}$ Resource Utilization

System Settings	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
Block PB: $g = 69$	24535	(97%)	7538	(14%)	44444	(87%)
NonBlock PB: $g = 59$	23751	(93%)	9130	(18%)	49761	(80%)
Block NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
Block MB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock MB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A

Table 4.45: $\mathbb{F}_{2^{409}}$ Execution Time (msec)

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
Block PB: $g = 69$	1.2873	3.6542	3.3202	3.1146	2.9898	3.1846
NonBlock PB: $g = 59$	1.1278	3.5558	3.2415	3.0489	2.9192	3.0742
Block NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
Block MB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock MB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A

Table 4.46: $\mathbb{F}_{2^{571}}$ Resource Utilization

System Settings	Slices	% Slices	FFs	% FFs	LUTs	% LUTs
Block PB: $g = 34$	24224	(95%)	9004	(17%)	43394	(85%)
NonBlock PB: $g = 29$	25278	(99%)	12556	(24%)	43246	(85%)
Block NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
Block MB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock MB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A

Table 4.47: $\mathbb{F}_{2^{571}}$ Execution Time (msec)

Param	Dbl-n-Add	RTNAF$_2$	RTNAF$_3$	RTNAF$_4$	RTNAF$_5$	RTNAF$_6$
Block PB: $g = 34$	2.7199	8.0784	7.3685	6.9312	6.6437	7.0387
NonBlock NB: $g = 29$	2.4966	7.9181	7.2475	6.8219	6.5285	6.8744
Block NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock NB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
Block MB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A
NonBlock MB: $g =?, w =?$	N/A	N/A	N/A	N/A	N/A	N/A

Chapter 5

Conclusion

The conclusions to the work done in this thesis has been divided into three sections. First, a number of implementation issues which had adverse effects on the results presented in previous chapters are outlined and discussed. Second, results from other relevant research is presented and compared to the results obtained from this work. Finally, based on the work done in this thesis, and the results from recent relevant research, such as [15], [9], [8] and [21], the possibilities of further research are outlined and discussed.

5.1 Implementation Issues

Throughout the work on this thesis numerous issues were encountered, as is to be expected when undertaking any technical work. However, a few of these issues deserve special attention as they had a significant impact on the results obtained in this thesis. This section will outline the most prominent issues and attempt to explain the underlying factors and how they could have possibly been mitigated.

First of all, the computational cost of RTNAF$_w$ encoding was much more significant than initially expected. Implementing this algorithm in the PPC core turned out to be a bad design choice. Even if the design were altered to run the PPC at its maximum frequency of 300 MHz the recoding operation would still be a significant addition to the runtime of scalar multiplication. The underlying issue is that RTNAF$_w$ recoding is an inherently sequential algorithm that requires large number arithmetic, and a simple processor that is clocked at a

low frequency is not ideal for such a situation. There is the possibility that implementing arithmetic units that have very large operands (greater than 128 bits) in hardware would have posed a significant advantage, however, exploring this is outside the scope of this thesis. Overall, the poor performance of the scalar recoding algorithm was a significant detriment to the scalar multiplication units and it is obvious why other works, such as [14], chose to assume scalar multiplication would be performed by a separate entity independent of their unit. Table 5.1 shows the recoding times for each RTNAF$_w$ width over the five bases. The percentage value shown under each time value is the percentage of overall runtime required by the recoding process for the fastest implementation of that particular field and recoding width.

Table 5.1: RTNAF Conversion Times (msec)

Field	**RTNAF$_2$**	**RTNAF$_3$**	**RTNAF$_4$**	**RTNAF$_5$**	**RTNAF$_6$**
$\mathbb{F}_{2^{163}}$	0.6370 (86%)	0.5713 (87%)	0.5356 (87%)	0.5104 (86%)	0.5137 (84%)
$\mathbb{F}_{2^{233}}$	1.1501 (89%)	1.0378 (90%)	0.9751 (90%)	0.9316 (90%)	0.9647 (87%)
$\mathbb{F}_{2^{283}}$	1.7156 (91%)	1.5605 (91%)	1.4732 (92%)	1.4120 (92%)	1.4596 (89%)
$\mathbb{F}_{2^{409}}$	3.2991 (93%)	3.0016 (93%)	2.8312 (93%)	2.7140 (93%)	2.8137 (92%)
$\mathbb{F}_{2^{571}}$	7.4048 (94%)	6.7538 (93%)	6.3749 (93%)	6.1167 (94%)	6.3430 (92%)

Second, as was mentioned previously, while the overall benefits of LD projective coordinates were very significant, there were certain caveats to utilizing this particular style of projective points. The primary issue with LD coordinates is that a neutral element only exists in theory. There are no set of projective coordinates N that can be used as an input to the LD point addition formula such that $P + N = N$. Instead the situation of a neutral element must be be treated as a special case. While this leads to a faster scenario if a neutral element is present in a point addition, as an unnecessary point addition can be avoided, in the more common case this leads to a slower unit as a number of conditional statements must checked to be sure that a neutral element is not present.

Lastly, there were significant timing constraint issues that became apparent in the larger systems. All of the systems and their respective components were targeted to run with a 100 MHz clock, and each of them met this constraint individually. However, when the systems were assembled timing problems became apparent for certain larger digit width and larger basis systems. The normal basis and mixed basis approaches were the most susceptible to timing errors, leading to the belief that the normal basis squaring unit was the primary cause. If this turned out to be the case this would make sense, especially for the larger basis systems, as the normal basis squaring unit was primarily routing. If this issue had been discovered earlier in the design of this thesis more time could have been allotted to tracking down and fixing this issue.

5.2 Comparison Against the Work of Others

In order to give an accurate depiction of the results obtained from this thesis, the results must be compared against those of other research endeavors. First we must discuss what exactly will be compared and how. Most other research efforts only focus on the smallest of the NIST recommended Koblitz curves, $\mathbb{F}_{2^{163}}$ and, therefore, the fastest results obtained in this work will be compared to the execution times achieved by others. Table 5.2 shows the pertinent results obtained in this thesis and those from other relevent research. It should be noted that the execution times are marked as with or without RTNAF$_w$ recoding included. The reason for this is that certain research efforts excluded scalar recoding from their computations and, as was mentioned earlier, in this thesis RTNAF$_w$ was done in an admittedly inefficient manner and its runtime greatly overshadows the cost of the remaining scalar multiplication algorithm.

The first relevant undertaking to compare against is the work done by Lutz and Hasan in [15]. A $\mathbb{F}_{2^{163}}$ Koblitz curve scalar multiplication hardware accelerator was created in [15] and targeted at a Xilinx XCV2000E FPGA running at 66 MHz. The final result produced a scalar multiple in 0.075 msec, using a strict polynomial basis approach with a polynomial

Table 5.2: Comparison Against the Work of Others for $\mathbb{F}_{2^{163}}$ (msec)

Source	Details	Runtime (msec)
This Research Xilinx V4 FX60	100 MHz RTNAF_5 NB, $g = 82, w = 163$	 (with recoding) 0.5902 (without recoding) 0.0798
Lutz and Hasan [15] Xilinx XCV2000E 2003	66 MHz TNAF_2 PB, $g = 41$	 (without recoding) 0.0750
Jarvinen and Skytta [9] Altera Stratix II EP2S180F1020C3 Sept. 2008	89 MHz & 156 MHz TNAF_2 NB, $g = 11$	 (with recoding) 0.0260
Jarvinen and Skytta [8] Altera Stratix II EP2S180F1020C3 April 2008	85 MHz & 185 MHz TNAF_4 NB, $g = 4$ & $g = 13$	 (with recoding) 0.0160
Sakiyama *et al.* [21] ASIC Design 2007	555.6 MHz TNAF_2 PB, $g = 12$	 (without recoding) 0.0120

basis multiplier having a digit width of $g = 41$. In comparison to this work, if the conversion process is ignored to match the situation presented by Lutz and Hasan, then the fastest unit is the normal basis approach ($g = 81$, $w = 163$) which requires 0.0798 msec to produce a scalar product. The work done in this thesis comes very close to the results produced by Lutz and Hasan, and it is most likely due to the advanced techniques (i.e. RTNAF_5) and FPGA (clocked at 100 MHz) that allowed the more flexible design presented in this thesis to be as close as it was to Lutz and Hasan's custom tuned FPGA implementation.

The next works to be compared are the latest from a series of research papers presented by Kimmo Jarvinen and Jorma Skytta. It should be noted that these two individuals were part of the research study for computing TNAF in hardware in [7], which was discussed earlier. In the latest research presented by these two a hardware accelerated elliptic curve scalar multiplication was created on a Stratix II, with focus placed on performing scalar multiplication on the $\mathbb{F}_{2^{163}}$ NIST recommended Koblitz curve. In [9] Jarvinen and Skytta take advantage of the independent nature of point addition when performing scalar multiplication on Koblitz curves and achieve an execution time of 0.026 msec. In this work

TNAF recoding (approximate to RTNAF$_2$) was implemented in hardware and the conversion process is part of the total execution time. This is obviously a much better achievement as compared to the results obtained in this thesis as the execution time was faster and a comparably less powerful FPGA was used. Next, in [8], Jarvinen and Skytta adapt the window method to their TNAF recoding hardware to achieve an execution time of 0.016 msec. This is an even more impressive result which shows the affects of their improved design choices.

Many things can be learned from the work done in [9] and [8]. First of all, in the scenario of performing scalar multiplication on Koblitz curves, there is the possibility that the use of a hybrid FPGA is overkill. RTNAF$_w$ is computationally too expensive to be implemented solely in the PPC and using the PPC as a complex state machine may be unnecessary as the algorithm for point multiplication on a Koblitz curves in not extremely complicated. Jarvinen and Skytta opted to use a custom state machine with instructions stored in ROM, which resulted in a fast system that still maintained flexibility. Of course this offers little viewpoint on the ease of implementation and debugging that using a hybrid FPGA offered, however, this brings us to the classic tradeoff between development time and execution time of a final product. Second, Jarvinen and Skytta have shown that extremely efficient systems can be produced even when not utilizing the ideal RTNAF$_w$ encoding proposed by Solinas. It is still unclear as to whether a system could be developed that utilized RTNAF$_w$ encoding efficiently, but Jarvinen and Skytta have shown that the loss of using a less efficient recoding may not be very significant. Lastly, and perhaps most importantly, Jarvinen and Skytta have shown that it is more efficient to parallelize the scalar multiplication process and use multiple multipliers with mid-range digit widths as opposed to focusing resources on a single fast and large multiplier. This is in line with the results of this thesis as Figure 4.14 shows that the $\mathbb{F}_{163}$ systems with $g = 8$ or $g = 17$ were the most efficient for the three different approaches.

Lastly, there is the work performed by Sakiyama *et al.* in [21]. In this work Sakiyama *et al.* created an ASIC to perform scalar multiplication on the NIST recommended Koblitz curve over $\mathbb{F}_{2^{163}}$. The design could be clocked at up to 555.6 MHz and produced a scalar

multiple in 0.012 msec. Recoding was not done in this unit, and it was assumed that a TNAF recoding input would be provided. These results show how an ASIC perform in comparison to the other Koblitz curve related research endeavors. It is surprising to see that the execution time of this unit, while it is beyond that achieved by the units in this thesis, is only slightly greater than the FPGA unit created by Jarvinen and Skytta.

Overall, while the results in this thesis did come close in comparison to some full scale FPGA implementations, it was significantly outdone by other research. Many things were learned from completing this work, and the final results obtained support that different design choices, or research into further options, could have resulted in a faster and more efficient system.

5.3 Ideas for Further Work

The work done in this thesis did fulfill the goals that were initially proposed, however, in light of the final results and the results obtained from recent related research, there are a number of areas which could be further explored. First of all, the utilization of a more advanced hybrid FPGA, such as a Virtex-5 FX model, would allow for the design of an improved system that could potentially run at a higher clock frequency. Table 5.3 shows the results of implementing the current multipliers on the Virtex-5 FX130. The additional resources and new technologies available on this chip could allow for the research of more advanced techniques and the synthesis of larger scalar multiplication systems. However, as the discussion in the previous section has shown, simply porting the architecture developed in this thesis to a more powerful platform would most likely not offer any significant gains. In addition to utilizing a more powerful system the methods and architectures presented in this work would need to be reworked in accordance to the results of this thesis and the results of current relevant research on Koblitz curves.

Second, as was mentioned earlier, the normal basis squaring unit may have benefited

Table 5.3: Synthesis Results for $\mathbb{F}_{2^{163}}$ on Virtex-5 FX130

Multiplier	Slice LUTs	Slice Registers
NB $g = 1$	840 (1%)	664 (0%)
PB $g = 1$	256 (0%)	501 (0%)
NB $g = 17$	3639 (4%)	754 (0%)
PB $g = 17$	2063 (2%)	509 (0%)
NB $g = 41$	7616 (9%)	821 (1%)
PB $g = 41$	4500 (5%)	503 (0%)
NB $g = 163$	26001 (31%)	763 (0%)
PB $g = 163$	14375 (17%)	662 (0%)

from a different implementation. Examining the tradeoff between execution time and resource utilization for a normal basis squaring unit that required more than a single clock cycle to compute sequential squarings could aid in increasing the efficiency of the normal basis approach. Also, creating a unit that could dynamically take advantage of reverse rotation could prove profitable. What is meant by this is that when an element is to be squared more than $m/2$ times consecutively, the vector could actually be rotated to the left instead of right as the operation would be equivalent and would possibly require fewer clock cycles or resources.

Third, this work and the work of Jarvinen and Skytta both show that it is worth it to investigate a way to utilize numerous smaller scalar multiplication units in parallel, as opposed to increasing the parameter settings on a single unit. It would be particularly interesting to study this in the context of a hybrid FPGA, such as the FX60, where multiple embedded processors are available.

Lastly, it would be interesting to explore other options for performing RTNAF$_w$ recoding. One such option would be a more in depth study and comparison between the RTNAF$_w$ recoding ideas presented by Solinas and the TNAF recoding ideas presented by Lutz, which were utilized by Jarvinen and Skytta. Theoretically the approach offered by Lutz should suffer from not using the reduction steps outlined by Solinas in [22], however, the work done by Jarvinen and Skytta seems to show that the overall effect is minimal. It would be interesting to explore how significant the effect of not using the reduction steps

proposed by Solinas actually are and if methods could be developed to aid in making the RTNAF$_w$ method a competitive solution with a primarily hardware based system.

Bibliography

[1] Powerpc 405 processor block reference guide. Xilinx, Inc., San Jose, CA, Datasheet UG018, May 2008.

[2] Roberto Avanzi, Clemens Heuberger, and Helmut Prodinger. Scalar multiplication on Koblitz curves using the Frobenius endomorphism and its combination with point halving: Extensions and mathematical analysis. *Algorithmica*, 46:249–270, 2006.

[3] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management part 1: General (revised). *NIST Special Publication 800-57*, 2006.

[4] Scott Vanstone Darrel Hankerson, Alfred Menezes. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., 2004.

[5] Darrel Hankerson, Julio López Hernandez, and Alfred Menezes. Software implementation of elliptic curve cryptography over binary fields. *Lecture Notes in Computer Science*, 1965, 2001.

[6] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Information and Computation*, 78:171–177, 1988.

[7] Kimmo Jarvinen, Juha Forsten, and Jorma Skytta. Efficient circuitry for computing τ-adic non-adjacent form. pages 232–235, 10-13 Dec. 2006.

[8] Kimmo Jarvinen and Jorma Skytta. High-speed elliptic curve cryptography accelerator for Koblitz curves. pages 109–118, April 2008.

[9] Kimmo Jarvinen and Jorma Skytta. On parallelization of high-speed processors for elliptic curve cryptography. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(9):1162–1175, Sept. 2008.

[10] Glenn Ramsey Jr. Hardware/software optimizations for elliptic curve scalar multiplication on hybrid FPGAs. Master's thesis, Rochester Institute of Technology, June 2008.

[11] Neal Koblitz. Cm-curves with good cryptographic properties. In *CRYPTO '91: Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, pages 279–287, London, UK, 1992. Springer-Verlag.

[12] Ming Li, Baodong Qin, Fanyu Kong, and Daxing Li. Wide-w-NAF method for scalar multiplication on Koblitz curves. *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, 2:143–148, July 30 2007-Aug. 1 2007.

[13] Julio Lopez and Ricardo Dahab. Improved algorithms for elliptic curve arithmetic in GF(2^n). In *Selected Areas in Cryptography*, pages 201–212, 1998.

[14] Jonathan Lutz. High performance elliptic curve cryptographic co-processor. Master's thesis, University of Waterloo, 2003.

[15] Jonathan Lutz and Anwarul Hasan. High performance FPGA based elliptic curve cryptographic co-processor. volume 2, pages 486–492 Vol.2, 5-7 April 2004.

[16] Nele Mentens, Siddika Berna Ors, and Bart Preneel. An FPGA implementation of an elliptic curve processor GF(2^m). In *GLSVLSI '04: Proceedings of the 14th ACM Great Lakes symposium on VLSI*, pages 454–457, New York, NY, USA, 2004. ACM.

[17] U.S. Department of Commerce, National Institute of Standards, and Technology. Digital signature standard (dss). Technical report, Federal Information Processing Standards Publication FIPS PUB 186-2, January 2000.

[18] A. Reyhani-Masoleh and M.A. Hasan. Low complexity bit parallel architectures for polynomial basis multiplication over GF(2^m). *Computers, IEEE Transactions on*, 53(8):945–959, Aug. 2004.

[19] Arash Reyhani-Masoleh. Efficient algorithms and architectures for field multiplication using gaussian normal bases. *IEEE Transactions on Computers*, 55(1):34–47, 2006.

[20] F. Rodriguez-Henriquez, N. Cruz-Cortes, and N.A. Saqib. A fast implementation of multiplicative inversion over GF(2^m). *Information Technology: Coding and Computing, 2005. ITCC 2005. International Conference on*, 1:574–579 Vol. 1, April 2005.

[21] Kazuo Sakiyama, Lejla Batina, Bart Preneel, and Ingrid Verbauwhede. High-performance public-key cryptoprocessor for wireless mobile applications. *Mob. Netw. Appl.*, 12(4):245–258, 2007.

[22] Jerome A. Solinas. An improved algorithm for arithmetic on a family of elliptic curves. *Lecture Notes in Computer Science*, 1294:357–426, 1997.

[23] Camille Vuillaume, Katsuyuki Okeya, and Tsuyoshi Takagi. Short-memory scalar multiplication for Koblitz curves. *IEEE Transactions on Computers*, 57(4):481–489, April 2008.